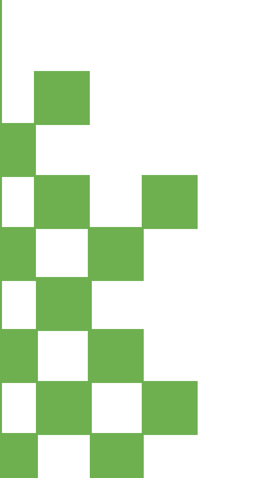
全国高等院校应用型**创新规划**教材 计算机系列

Python 程序设计实用教程

杨连贺 董禹龙 房 超 主 编
毕璐琪 梁润宇 杨 阳 彭进香 副主编



+

+

+


- 以基础理论—实用技术—实训为主线
- 按照教与学的实际需要取材谋篇
- 精心设置“小型案例实训”，旨在培养学生的实践能力
- 配备免费教学资源——教学课件、程序源代码、授课计划及习题答案等

清华大学出版社

全国高等院校应用型创新规划教材·计算机系列

Python 程序设计实用教程

杨连贺　董禹龙　房　超　主　编
毕璐琪　梁润宇　杨　阳　彭进香　副主编

清华大学出版社
北　京

内 容 简 介

Python 是一门简单易学、功能强大的编程语言，它内建了高效的数据结构，能够用简单而又高效的方式编程。它优雅的语法和动态的类型，再结合它的解释性，使其成为在大多数平台下编写脚本或开发应用程序的理想语言。

本书系统而全面地介绍了 Python 语言的全部内容，既能为初学者夯实基础，又适合程序员提升技能。考虑到近几年数据挖掘技术和网络编程技术的发展，本书加入了 Python 语言在科学计算、网络编程、并发技术和数据可视化方面的内容。与一般的 Python 语言教材相比，本书增加了许多实际案例的应用，可以让读者更好地将 Python 基础知识应用到实际工作中。书中的每道例题，均以屏幕截图的方式原滋原味地给出运行结果，便于读者分析程序。

本书可作为高等院校各专业的 Python 语言教材，亦可作为软件开发人员的参考资料，还可作为读者自学 Python 语言的参考书。

图书在版编目(CIP)数据

Python 程序设计实用教程/杨连贺，董禹龙，房超主编. —北京：清华大学出版社，2018
(全国高等院校应用型创新规划教材·计算机系列)
ISBN 978-7-302-50047-6

Ⅰ. ①P…　Ⅱ. ①杨…　②董…　③房…　Ⅲ. ①软件工具—程序设计—高等学校—教材　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 086665 号

责任编辑： 汤涌涛
封面设计： 杨玉兰
责任校对： 宋延清
责任印制： 丛怀宇
出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载： http://www.tup.com.cn, 010-62791865
印 装 者： 北京国马印刷厂
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 23.5　　**字　　数：** 572 千字
版　　次： 2018 年 6 月第 1 版　　**印　　次：** 2018 年 6 月第 1 次印刷
印　　数： 1～2500
定　　价： 56.00 元

产品编号：074820-01

前　　言

根据 TIOBE 网站的最新排名，Python 已超越 C#，与 Java、C、C++一起，成为全球前四大流行语言。IEEE 发布的 2017 年编程语言排行榜则将 Python 排在榜首。

Python 也是美国大学选用最多的语言，著名的哈佛大学、麻省理工学院、加州大学伯克利分校、卡耐基•梅隆大学等，已将 Python 语言作为计算机专业和非计算机专业的入门语言。Python 崇尚简、短、精、小，其应用几乎无限制，各方面地位超然。Python 在软件质量控制、提升开发效率/可移植性、组件集成、丰富的库支持等方面，均处于先进地位。更重要的是，Python 简单易学、免费开源、可移植、可扩展、可嵌入。此外，Python 还支持面向对象，而且它的面向对象甚至比 Java 和 C#.NET 更彻底。

Python 是高“性价比”的语言。它合理地结合了高性能与低成本(代码量小、维护成本低、编程效率高)的特色，致力于用最简洁、最简短的代码完成任务。

完成同样的业务逻辑时，在其他编程语言中可能需要编写大量的代码，而在 Python 中只需要调用内建函数或内建对象的方法即可实现，甚至可以直接调用第三方扩展库来完成。一般情况下，Python 的代码量仅仅是 Java 的 1/5，足见 Python 编程的高效。

Python 是应用“无限制”的语言。它被广泛应用于后端开发、游戏开发、网站开发、科学计算、大数据分析、云计算、图形开发等领域。美国中央情报局 CIA 网站、世界上最大的视频网站 YouTube、国内最大的问答社区“知乎”等，都是用 Python 开发的，搜狐、金山、腾讯、盛大、网易、百度、阿里、淘宝、土豆、新浪、果壳等著名的 IT 公司都在使用 Python 完成各种各样的任务。

Python 是一种代表“简单主义”思想的语言。它的设计哲学是优雅、明确、简单。阅读一个良好的 Python 程序，感觉就像是在阅读英语，尽管这个英语的要求非常严格！Python 的这种伪代码本质，是它最大的优点之一。

Python 是“高层次”的语言。它内建优异的数据结构，很容易表达各种常见的数据结构，不再需要定义指针、分配内存，编程也简单了许多，也无须考虑程序对内存的使用等底层细节，把许多机器层面上的细节隐藏起来，凸显出逻辑层面的编程思考。

Python 是免费、开源、跨平台的高级动态编程语言。它支持命令式编程、函数式编程，全面支持面向对象编程；它语法简洁、清晰，拥有功能丰富而强大的标准库和大量的第三方扩展库；它使用户能够专注于解决问题，而不是去搞明白语言本身，这是它开发效率高的根本原因。

由此可见，用“出类拔萃”来形容 Python 并不为过。Python 以如此众多的优势，吸引着无数程序员投身于其中。网上的一句流行语颇耐人寻味：“人生苦短，我用 Python”。

在国外，“Python 热”正在逐步升温，涉及方方面面的领域；在国内，越来越多的大学已将 Python 列入本科生的必修课程或选修课程，越来越多的 IT 企业将开发语言瞄向了 Python。可以预见的是，国内的“Python 热”即将掀起，本书的出版迎合了这一趋势。

本书的内容组织说明如下。

为了拓展应用范围，充分利用现有资源，对于 Python 程序员而言，熟练运用第三方扩展库是非常重要的。使用成熟的扩展库，可以帮助我们快速地实现业务逻辑，达到事半功倍的效果。但是，第三方扩展库的理解和运用，无疑要建立在对 Python 基础知识和基本数据结构熟练掌握的基础上。因此，本书兼顾“基础”与“应用”两个方面，前 6 章把重点放在基础上，通过大量的经典例题，讲解 Python 语言的核心内容；后 6 章则把重点放在应用上，通过大量的案例，介绍 Python 在实际开发中的应用。关于不同应用领域的第三方扩展库，读者可以参考附录 B，并结合自己的专业领域查阅相关文档。

本书共分 12 章，主要内容组织如下。

第 1 章：Python 程序设计入门。介绍什么是 Python，学习 Python 的原因，Python 的发展历史，多种平台下 Python 环境的搭建，使用集成开发环境 IDLE 来帮助学习 Python，Python 常用的开发工具，最后给出本书的第一个 Python 程序。

第 2 章：Python 语言基础。讲解 Python 的语法和句法，Python 的数据类型，Python 的常量与变量，Python 的运算符与优先级，Python 的数值类型，Python 的字符串类型，Python 的高级数据类型(列表、元组、字典、集合)，最后介绍正则表达式及其应用。

第 3 章：Python 流程控制。讲解 if 语句和 for 语句的基本格式、执行规则、嵌套用法，range()函数在循环中的使用方法，while 语句的基本格式、执行规则、嵌套用法，最后介绍 break、continue、pass 等关键字在循环中的使用方法。

第 4 章：函数模块。讲解 Python 代码编写规范和风格，函数的定义与调用，函数参数的传递，Python 变量作用域，函数与递归，迭代器与生成器，Python 自定义模块，输入输出语句的基本格式及执行规则，匿名函数的定义与使用。

第 5 章：文件与异常处理。介绍文件和文件对象，讲解基于 os 模块的文件操作方法，基于 shutil 模块的文件操作方法，文本文件、CSV 文件、Excel 文件的基本操作，HTML、XML 文档的基本操作，最后介绍 Python 的异常处理机制及 Python 程序的调试方法。

第 6 章：面向对象编程。介绍面向对象技术，讲解类与对象的定义和使用，类的属性与方法，类的作用域与命名空间，类的单继承和多继承，最后以数个典型实例讲解面向对象程序设计的应用。

第 7 章：数据库编程。讲解数据库技术基础，SQLite 和 MySQL 数据库的数据类型、基本操作，使用 Python 操作 SQLite 和 MySQL 数据库的方法。

第 8 章：Web 开发。讲解 Web 应用的工作方式，MVC 设计模式，CGI 通用网关接口，使用模板快速生成 Web 页面。

第 9 章：使用 Python 进行数据分析。讲解使用 Python 进行数据挖掘的原因，介绍 NumPy 库、SciPy 库、Matplotlib 库和 Pandas 库，最后通过数理统计中的数据离散度分析和数据挖掘中的离群点分析等典型案例，介绍 Python 在数据可视化方面的应用。

第 10 章：GUI 编程和用户界面。讲解 GUI 界面的概念，Tkinter 模块及其各种组件，网格布局管理器，最后介绍 GUI 编程。

第 11 章：多进程与多线程。介绍多进程与多线程的概念，讲解多进程与多线程的区别，进程间通信技术，进程池，最后介绍 thread 锁。

第 12 章：网络编程。讲解计算机网络基础知识，Socket 通信技术，urllib 库及其使用，端口扫描器，最后以一个简单的网络爬虫为例，对前几章的知识进行综合应用。

本书最大的特点是内容紧凑、案例丰富、学以致用；程序输出原滋原味，既有正确输出的结果，又有错误输出的提示，让读者既能从“正”的方面学到经验，又能从“负”的方面吸取教训，使经验与教训兼而得之。全书总体内容按照先基础、后应用的顺序安排。前 6 章为基础篇，其内容循序渐进；后 6 章为应用篇，其内容自成体系；每个知识点按照先讲解知识、后给出案例的顺序编写；每个软件都配有安装过程截图，每道例题都配有运行结果截图，一目了然。

本书作者具有近 30 年的程序设计教学经验，讲授过多门编程语言课程，并编写过大量的应用程序，青年时期曾参加过市级讲课大赛并取得优异成绩，特别是在美国访学期间，用 Python 语言开发过较大规模的软件。在内容的组织和安排上，本书结合了作者多年教学与科研中积累的经验，并巧妙地将其糅合到相应的章节中。

本书以目前流行的 Python 3 为基础，适当兼顾 Python 2.x，既讲解 Python 的基础知识，又适当介绍 Python 在各个方面的应用，因而，可以满足不同层次读者的需要。

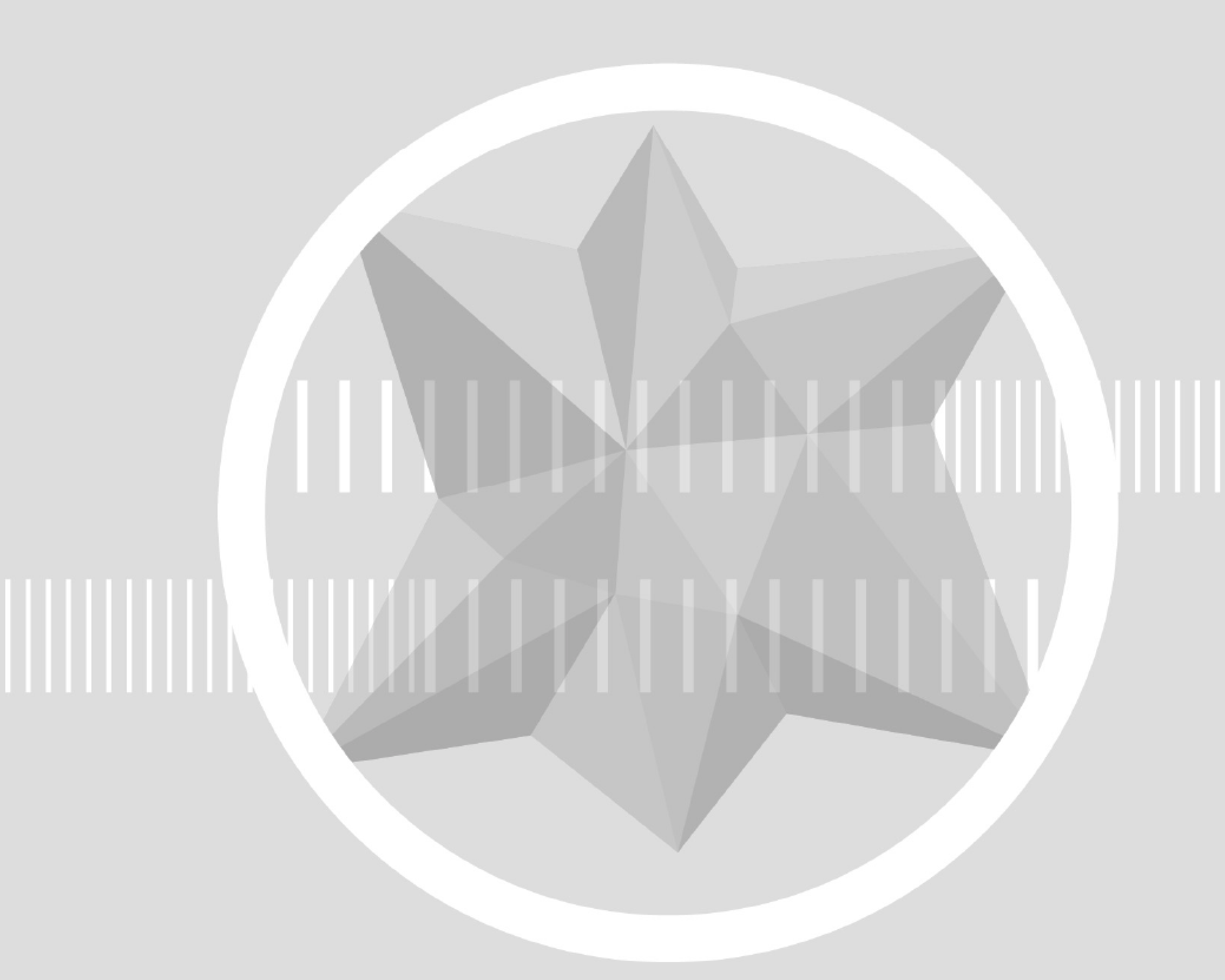
本书可以作为高等院校计算机或非计算机专业程序设计语言公共课或选修课教材，基础教学建议选取前 6 章内容，推荐 36 学时；“基础+应用”教学建议按“6+n”方式选取教学内容，后面 6 章可根据专业需要择其一二，或全部选用，推荐 42~64 学时。建议采用边讲边练的教学模式。本书可以作为具有一定 Python 基础的读者进一步学习的资料，可供参加各类计算机考试的人员学习和参考，也可以作为从事数据分析、网络运维、数据库开发、Web 开发、界面设计、软件开发等工作的工程师的参考资料。对于打算利用业余时间快乐地学习一门编程语言并编写一些小程序来自我娱乐的读者，本书是首选的学习资料。本书亦适合对编程有着浓厚兴趣的中小学生作为课外阅读资料。

本书由天津工业大学杨连贺、董禹龙、房超主编，该校毕璐琪、梁润宇及天津市电子计算机研究所杨阳、湖南应用技术学院彭进香为副主编。限于作者的经验和水平，书中的错误与不足之处在所难免，希望得到专家和读者的批评指正。

本书编写过程中，天津工业大学计算机科学与软件学院硕士研究生张海潮和焦翠姣在程序调试方面做了很多工作，在此一并向她们表示衷心的感谢。

作　者
2018 年 5 月于天津工业大学

目　录



第 1 章

Python 程序设计入门

本章要点

(1) Python 概述。
(2) 什么是 Python?
(3) 为什么学 Python?
(4) Python 的发展。
(5) Windows 下 Python 环境的搭建。
(6) Linux 下 Python 环境的搭建。
(7) 使用 IDLE 来帮助学习 Python。
(8) Python 常用的开发工具。
(9) 第一个 Python 程序。

学习目标

(1) 了解 Python 语言。
(2) 理解学习 Python 语言的目的。
(3) 了解 Python 语言的发展。
(4) 熟悉常用平台下的 Python 开发环境及其搭建。
(5) 掌握 IDLE 的使用方法。
(6) 了解其他的常用开发工具。

本章向读者引入在国内方兴未艾的一门高级程序设计语言——Python。常用的编程语言达数十种之多，其功能各有千秋，应用领域也大相径庭。Python 之所以能够从众多的高级语言中脱颖而出，是因为 Python 是一种代表简单主义思想的语言，但集功能广泛与强大于一身。同样的功能，只用很少的代码就可以实现，编写的程序清晰易懂，优雅美观。用“高效开发+简单易学”来形容 Python 是恰如其分的。

1.1 Python 概述

Python 是一种简单易学、功能强大的编程语言，它继承了传统编译语言的强大性和通用性，具有高层次的数据结构，支持面向对象的编程方法。

Python 优雅的语法、动态的类型，连同它天然的解释性，使其成为在大多数平台下进行许多领域快速应用开发的理想语言。

1.1.1 什么是 Python

Python 是一门跨平台的、开源的、免费的、解释型高级动态编程语言，是由荷兰人 Guido van Rossum 在 1989 年始创的，其名字来源于一个喜剧。Python 的第一个公开发行版是在 1991 年初发布的，历经 20 余年的发展，目前最高版本为 3.6。根据 TIOBE 的最新排名，Python 已超越 C#，与 Java、C、C++一起，成为全球前四大最流行的语言。

也许 Guido van Rossum 最初并没有想到今天 Python 会在各种行业中获得如此广泛的

使用。著名的自由软件作者 Eric Raymond 在“如何成为一名黑客”一文中，将 Python 列为黑客应当学习的四种编程语言之一，并建议人们从 Python 开始学习编程。这的确是一个非常中肯的建议，对于那些从未学过编程的人，或者对非计算机专业的编程学习者而言，Python 不失为最好的选择之一。欧美的很多大学(如 MIT、Stanford 等)都以 Python 作为入门语言，之后再学习 C/C++，甚至很多非计算机专业的学生也开设此课。相信在不久的将来，国内高校将兴起一股“Python 热”。

有些人喜欢用“胶水语言”来形容 Python，是因为它可以很轻松地把许多其他语言编写的模块结合在一起。Python 具有丰富和强大的基本类库，能够把用其他语言制作的各种模块(尤其是 C/C++)很轻松地联结在一起。常见的一种应用情形是，使用 Python 快速生成程序的原型(有时甚至是程序的最终界面)，然后对其中有特别要求的部分，用更合适的语言改写，比如 3D 游戏中的图形渲染模块，性能要求非常高，就可以用 C/C++重写，而后封装为 Python 可以调用的扩展类库。

很多初学 Java 的人都会被 Java 的 CLASSPATH 搞得晕头转向，花上大半天才搞清楚原来是 CLASSPATH 搞错了，以至于连自己的“Hello World!”程序都无法运行。而用 Python，就不会出现此类问题，因为 Python 是一种解释型语言，同时，也是一种脚本语言，写好代码即可直接运行，省去了编译、链接的一系列麻烦，对于需要多动手实践的初学者而言，减少了很多出错的机会。不仅如此，Python 还支持交互的操作方式，如果只是运行一段简单的小程序，连编辑器都可以省略，直接输入即可运行。

谈及“解释型”和“脚本语言”，人们常常会有一种担心：解释型语言通常很慢。

的确，从运行速度来讲，解释型语言通常会慢一些，但 Python 的速度却比人们想象的快很多。虽然 Python 是一种解释型语言，但实际上也可以编译(就像编译 Java 程序一样)，即把 Python 程序编译为一种特殊的 ByteCode。程序运行时，执行的实际上是 ByteCode，省去了对程序文本的分析解释，速度自然得到了显著的提升。在用 Java 编程时，人们往往崇尚一种 Pure Java 方式，除了虚拟机外，一切代码都用 Java 编写，无论是基本的数据结构还是图形用户界面，而 Pure Java 的 Swing 却成为无数 Java 应用开发者的噩梦。

Python 与之不同，它崇尚的是实用，它的整体环境是用 C 编写的，其中的很多基本功能和扩展模块均用 C/C++编写，执行这一部分代码时，其速度就是 C 的速度。用 Python 编写的普通桌面应用程序，其启动、运行的速度与用 C 书写的程序相差无几。除此之外，通过一些第三方软件包，用 Python 编写的源代码还可以以类似于 JIT 的方式运行。此举可大大提高 Python 代码的运行速度，针对不同类型的代码，运行速度将有 2~100 倍的提升。

Python 是一种结构清晰的编程语言，使用缩进的方式来表示程序的嵌套关系，可谓是一种创举。此举把过去软性的编程风格升级为硬性的语法规定，再也不需要在不同的风格之间进行选择。与 Perl 语言不同，Python 中没有各种隐晦的缩写，也不需要强记各种符号的含义。用 Python 书写的程序很容易读懂，这是不少人的共识。虽然 Python 是一种面向对象的编程语言，但它的面向对象却不像 C++那样强调概念，而是更注重实用。说到底，Python 不是为了体现对概念的完整支持而把语言搞得很复杂，而是用最简单的方法，让编程者能够享受到面向对象带来的好处，这正是 Python 能够吸引众多支持者的原因之一。

Python 是一种功能丰富的语言，它拥有一个强大的基本类库，同时拥有数量众多的第三方扩展库，这使得 Python 程序员无须去羡慕 Java 的 JDK。Python 为程序员提供的基本

功能十分丰富，使得人们写程序时无需一切从底层做起。

C 语言用来编写操作系统等贴近硬件的系统软件，所以，C 语言适合开发那些追求运行速度、充分发挥硬件性能的程序，而 Python 是用来编写应用程序的高级编程语言。许多大型网站就是用 Python 开发的，例如 YouTube、Instagram，还有国内的豆瓣等。很多大型的组织机构，包括 Google、Yahoo 等，甚至 NASA(美国航空航天局)，都大量地使用 Python。除此之外，还有搜狐、金山、腾讯、盛大、网易、百度、阿里、淘宝、土豆、新浪、果壳等公司，都在使用 Python 完成各种各样的任务。

Guido van Rossum 给 Python 的定位是“优雅、明确、简单”，所以 Python 程序看上去总是简单易懂。初学者学 Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。

总地来说，Python 的哲学就是简单优雅，尽量写容易看明白的代码和少的代码。

1.1.2 为什么学 Python

之所以越来越多的人选择学习 Python，是因为 Python 有着与众不同的特性。

(1) 易用性与高速度的完美结合。

在为数众多的高级语言中，在易用性和速度上结合得最完美的非 Python 莫属。通过丧失一点点经常可以忽略不计的运行速度而获得更高的编程效率，这是众多的编程者选择 Python 的原因。

(2) 自动的垃圾回收机制。

Python 的自动垃圾回收机制是高级编程语言的一种基本特性，用支持这一功能的语言编程，程序员通常无须关心内存泄漏问题，而用 C/C++书写程序时，这却是最重要的需要认真考虑而又很容易出错的问题之一。

(3) 内建的优异数据结构。

数据结构是程序的重要组成部分。在用 C 编程时，对于链表、树、图这些数据结构，需要用指针仔细表达，而这些问题在 Python 中简单了很多。在 Python 中，最基本的数据结构是列表、元组和字典，用它们表达各种常见的数据结构可谓轻而易举。由于不再需要定义指针、分配内存，编程也简单了许多。

(4) 简易的 CORBA 绑定。

公用对象请求代理体系结构(Common Object Request Broker Architecture，CORBA)是一种高级的软件体系结构，它是与语言无关、与平台无关的。

C++、Java 等语言都有 CORBA 绑定，但与它们相比，Python 的 CORBA 绑定却容易了很多，因为在程序员看来，一个 CORBA 的类和 Python 的类用起来以及实现起来并没有什么差别。没有复杂体系结构的困扰，用 Python 编写 CORBA 程序也变得容易了。

(5) 成熟的跨平台技术。

随着 Linux 的不断成熟，越来越多的人转到 Linux 平台下工作，软件的开发者自然就希望自己编写的软件可以在所有平台下运行。Java 的“一次编写处处运行”口号曾使它成为跨平台开发工具的典范，但其运行速度却不被人们看好。

Python 不仅支持各种 Linux/Unix 系统，还支持 Windows，甚至在 Palm 上都可以运行 Python 程序。Python 不仅支持老一些的 TK，还支持新的 GTK+、QT 以及 wxWidget，而这些都可以在多个平台下工作。通过它们，程序员就可以编写出漂亮的跨平台图形用户界面(Graphical User Interface，GUI)程序。

(6)　高可扩展性与“乘坐快车”。

如果希望一段代码可以很快地执行，或者不希望公开一个算法，则可以使用 C/C++编写这段程序，然后在 Python 中调用，从而实现对 Python 程序的扩展。换言之，程序员可以用 C/C++为 Python 编写各种各样的模块，这不仅可以让程序员以 Python 的方式使用系统的各种服务和用 C/C++编写的优秀函数库与类库，还可以大幅度提高 Python 程序的运行速度。用 C/C++编写 Python 的模块并不复杂，而且为了简化这一工作，人们还制作了不少工具，用来协助这一工作。正因如此，现在各种常用的函数库和类库都有 Python 语言的绑定，用 Python 可以做到的事情越来越多了。

Python 从一开始就特别关注可扩展性。Python 可以在多个层次上扩展。从高层上，可以直接引入.py 文件，在底层则可以引用 C 语言的库。Python 程序员可以快速地使用 Python 写.py 文件作为扩展模块。但当运行速度作为重要的考虑因素时，Python 程序员可以深入底层，写 C 程序，编译为.so 文件后引入到 Python 中使用，就如同人们乘坐快车一样。Python 恰似使用钢来构建房子，规定好大的框架之后，程序员可以在此框架下相当自由地扩展或更改。

(7)　隐藏细节，凸显逻辑。

Python 将许多机器层面上的细节隐藏，交给编译器处理，并凸显出逻辑层面的编程思考。Python 程序员可以花更多的时间用于思考程序的逻辑，而不是具体的实现细节。这一特征吸引了广大的程序员。

Python 有如此众多的优点，读者是否会觉得 Python 是一把万能钥匙呢？答案自然是否定的。这是因为，Python 虽然功能强大，但它却不是万能的。不同的语言有不同的应用范围，想找一把“万能钥匙”是不太可能的。C 和汇编语言适合编写系统软件，如果用它们来编写企业应用程序，恐怕很难得心应手。因此，聪明的程序员总是选用合适的工具去开发软件。如果要编写操作系统或驱动程序，Python 很显然是做不到的。要写软件，没有哪个工具是万能的，现在之所以有那么多的编程语言，就是因为不同的语言适合做不同的事情。因此，选择适合自己的语言才是最重要的。

常言道：“好钢要用在刀刃上。”要想用有限的时间完成尽量多的任务，就要把各种无关的问题抛弃，而 Python 恰恰提供了这种方法，这也是我们要学习 Python 的原因。

1.1.3　Python 的发展

1989 年，为了打发圣诞节假期，Guido 开始写 Python 语言的编译器。Python 这个名字，来自 Guido 所挚爱的电视剧 Monty Python’s Flying Circus。他希望这个新的称作 Python 的语言能符合他的理想：创造一种介于 C 和 Shell 之间、功能全面、易学易用、可拓展的语言。

1991 年，第一个 Python 编译器诞生，它是用 C 语言实现的，并能够调用 C 语言的库

文件。Python 从一开始就具有了类、函数、异常处理，并且拥有包含表和字典在内的核心数据结构，以及以模块为基础的拓展系统。

1994 年 1 月，Python 1.0 推出，其中增加了 lambda、map、filter 和 reduce。

1999 年，Python 的 Web 框架之祖——Zope 1 发布。

2000 年 10 月 16 日，Python 2.0 问世，加入了内存回收机制，构成了现在 Python 语言框架的基础。

2004 年 11 月 30 日，Python 2.4 版出现；同年，目前最流行的 Web 框架 Django 诞生。

2006 年和 2008 年先后推出的 2.5 和 2.6 为过渡版本(2008 年已推出 3.0 版本)，2010 年推出的 2.7 版为 2.x 的最后一版。

2014 年 11 月，Python 2.7 将在 2020 年停止支持的消息被发布，并且不会再发布 2.8 版本，同时建议用户尽可能地迁移到 3.4+。

Python 最初发布时，在设计上有一些缺陷，比如 Unicode 标准晚于 Python 出现，所以一直以来，对 Unicode 的支持并不完全，而 ASCII 码支持的字符很有限。早期版本对中文的支持不好，Python 3 相对于 Python 早期的版本是一个较大的升级，Python 3 在设计的时候没有考虑向下兼容，所以很多早期版本的 Python 程序无法在 Python 3 下运行。为了照顾早期的版本，推出了过渡版本 2.6，该版基本上使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。2010 年继续推出了兼容版本 2.7，大量 Python 3 的特性被反向迁移到了 Python 2.7。2.7 比 2.6 进步非常多，同时拥有大量的 Python 3 中的特性和库，并且照顾了原有的 Python 开发人群。

前面刚刚提到，Python 2.7 是 2.x 系列的最后一个版本，已经停止开发，不再增加新的功能，而且将于 2020 年终止支持，这意味着所有最新的标准库更新改进，只会在 3.x 版本里出现。Guido 决定清理 Python 2.x，并且不再兼容旧版本。

2.0 版到 3.0 版最大的一个改变，就是使用 Unicode 作为默认编码。Python 2.x 中直接写中文会报错，Python 3 中可以直接写中文了。从开源项目来看，支持 Python 3 的比例已经显著提高，知名的项目一般都支持 Python 2.7 和 Python 3+。

Python 3 比 Python 2 更规范、统一，去掉了不必要的关键字。Python 3.x 还在持续改进，截至本书定稿，Python 的最高版本是 2017 年 7 月 17 日推出的 3.6.2 版。我们推荐大家使用 Python 3.x。

1.2 Python 开发环境的搭建

Python 可在多种平台下运行，但考虑到目前应用较多的是 Windows 平台和 Linux 平台，故本节分别介绍这两种平台下 Python 开发环境的搭建。

1.2.1 Windows 下 Python 开发环境的搭建

(1) 下载 Python 安装包。

首先登录 Python 官方网站 http://www.python.org/download/，点击 Download 后，选择适合自己版本的安装包，并下载之。

(2)　安装 Python。

双击下载的安装包，一路单击 Next，最后单击 Finish。

(3)　配置环境变量。

Python 安装完毕后，还需要把 Python 的安装目录添加到系统的 PATH 环境变量中。

(4)　测试。

在 DOS 命令提示窗口中输入“python”，若显示图 1-1 所示的界面，则说明 Python 已经安装成功。

```
C:\windows\system32\cmd.exe - python
C:\Python34>python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1-1　Python 安装成功的显示界面

(5)　演示 Python 命令。

按照很多资料介绍的，输入的第一个命令是“print ‘Hello world!’”，如图 1-2 所示。

结果显示“SyntaxError”(句法错误)。为什么会出错呢？原因在于，很多资料都是以 Python 2.x 为背景介绍的，而现在我们安装的是 3.x 版本，3.x 要求采用的写法是 print (‘Hello World!’)，即把 print 视为一个函数，而不是命令(考虑到习惯，本书仍称其为 print 命令)，因而需要使用括号，并把字符串放在引号中(单引号、双引号均可，但首尾必须相同)。改正后的运行结果如图 1-3 所示。

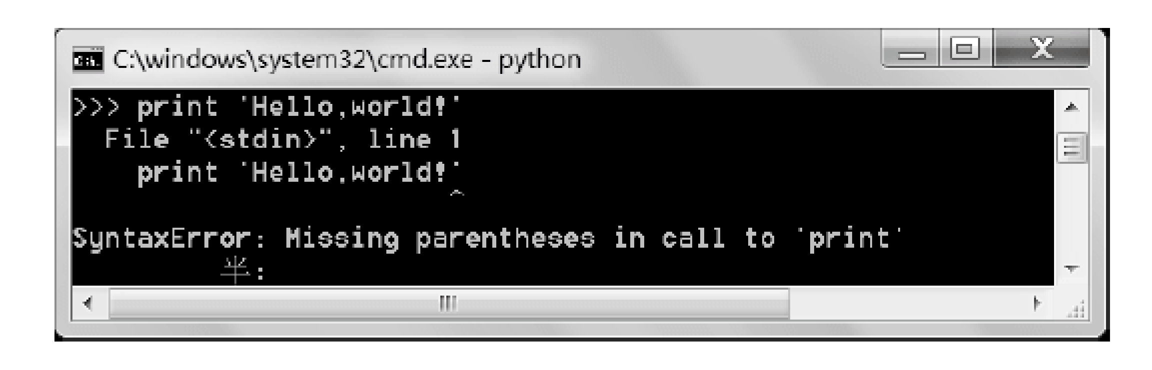


图 1-2　想要显示“Hello world!”

```
C:\windows\system32\cmd.exe - python
>>> print('Hello,world!')
Hello,world!
>>> _
```

图 1-3　命令行上显示“Hello world!”

(6)　集成开发环境的使用。

经过以上测试可知，Python 环境已安装完毕，但如何开发软件呢，难道用这种命令行方式开发？当然不是，因为有众多的集成开发环境(Integration Development Environment，IDE)可供使用。

在 Windows 下安装 Python 的同时，也默认安装了其自带的集成开发环境 IDLE，其启动方法是：开始→所有程序→Python 3.4→IDLE，启动后的界面如图 1-4 所示。

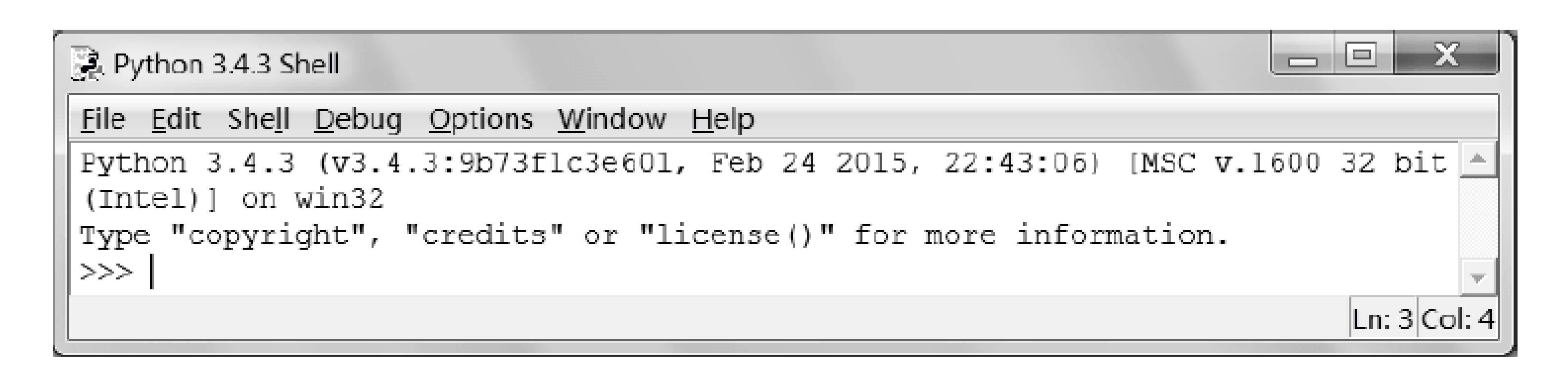


图 1-4　Python 的集成开发环境 IDLE 界面

当然，Python 的集成开发环境远不止 IDLE，常用的还有如下几种。

(1) Eclipse：大名鼎鼎的 Eclipse，此处不予介绍。

(2) Komodo Edit：一个免费的、开源的、专业的 Python IDE，其特征是非菜单的操作方式，开发效率高。

(3) Vim：一个简洁、高效的工具，也适合做 Python 开发。

(4) SublimeText 也是适合 Python 开发的 IDE 工具，SublimeText 虽然仅仅是一个编辑器，但是它有丰富的插件，使得对 Python 开发的支持非常到位。

(5) Pycharm：一个跨平台的 Python 开发工具，是 JetBrains 公司的产品。其特征包括：代码自动完成、集成的 Python 调试器、括号自动匹配、代码折叠。Pycharm 支持 Windows、MacOS 以及 Linux 等系统，而且可以远程开发、调试、运行程序。尽管这个 IDE 功能确实很强大，也很好用，但使用一段时间后要付费。

1.2.2 Linux 下 Python 开发环境的搭建

前一小节介绍的是 Windows 平台下 Python IDLE 安装和调试的过程。通常的 Linux 系统，如 Ubuntu、CentOS 等，都已经默认地随系统安装好 Python 程序了，只不过在 Linux 类系统中，这个 IDLE 被称为 Python 解释器，它是从终端模拟器中输入“python”这个命令启动的。Python 编程的一切工作都是从这个 IDLE 编辑器开始的。在入门后，可以选择更多自己喜欢的 Python 编辑器，如专业级 Python 编辑器 WingIDE。

遗憾的是，早期版本 Linux 操作系统中内置的 Python 版本多是 2.x 的，想在 3.x 下开发软件，必须安装 3.x 版的 Python。现以 Ubuntu(一个以桌面应用为主的 Linux 操作系统)为例，简要介绍一下 Linux 下 Python 3 开发环境的搭建过程。

(1) 安装 Python 3。

Ubuntu 自身是安装 Python 2 的，例如，在 Ubuntu 16.04 中安装的就是 Python 2.7。但我们需要在 Python 3 环境下进行软件开发，所以必须安装 Python 3。不过由于 Ubuntu 很多底层采用了 Python 2，所以安装 Python 3 时不能卸载 Python 2。首先执行以下各行命令：

```
sudo cp /usr/bin/python /usr/bin/python bak
sudo rm /usr/bin/python
sudo ln -s /usr/bin/python3.5 /usr/bin/python
```

然后输入“python”，检查版本是否正确(版本应该为 Python 3.5)。

(2) 安装 Sublime Text 3。

俗话说：“工欲善其事，必先利其器。”在进行 Python 开发时，一定要先选择一个好的编辑器。Sublime Text 3(ST3)是一款轻量级、跨平台的文本编辑器，可以安装在 Ubuntu、Windows 和 Mac OS X 上，有一个专有的许可证，但该程序也可以免费使用。如果想拥有更高级的版本，可付费获取。

打开终端，输入下列命令：

```
sudo add-apt-repository ppa:webupd8team/sublime-text-3
sudo apt-get update
sudo apt-get install sublime-text-installer
```

卸载 sublime text 命令：

```
sudo apt-get remove sublime-text-installer
```

这时候会发现桌面上并没有出现 Sublime Text 3，那应当如何打开它呢？从终端输入 subl 才能启动 sublime-text，启动后，直接将它锁定到侧边栏即可。

(3)　配置 Sublime Text 3。

为了使用众多的插件来扩展 Sublime 的功能，需要安装一个称为 Package Control 的插件管理器——这个东西必须手动安装。但是一旦安装好，就可以使用 Package Control 来安装、移除或者升级所有的 ST3 插件了。

按“Ctrl + ~”组合键打开控制台，在控制台里输入以下代码：

```
import urllib.request,os;pf='Package Control.sublime-package';ipp=
sublime.installed packages path();urllib.request.install opener(urllib.
request.build opener(urllib.request.ProxyHandler()));open(os.path.join
(ipp,pf),'wb').write(urllib.request.urlopen('http://sublime.wbond.net/'
+pf.replace(' ','%20')).read())
```

输完以后，按 Enter 键就可以执行了。

现在可以通过快捷键 Cmd+Shift+P 打开 Package Control 来安装其他的插件了。输入 install 后，就能看见屏幕上出现了 Package Control: Install Package(如图 1-5 所示)，按回车键，然后搜索想要的插件，想装哪个插件，直接点击即可。

因为我们想要配置 Python，所以选择第一个，选择后会出现一个文本框，如图 1-6 所示。

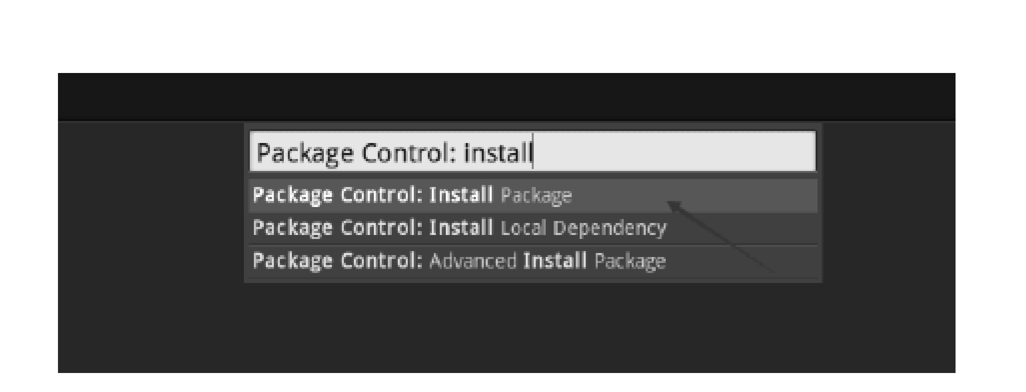


图 1-5　插件的安装

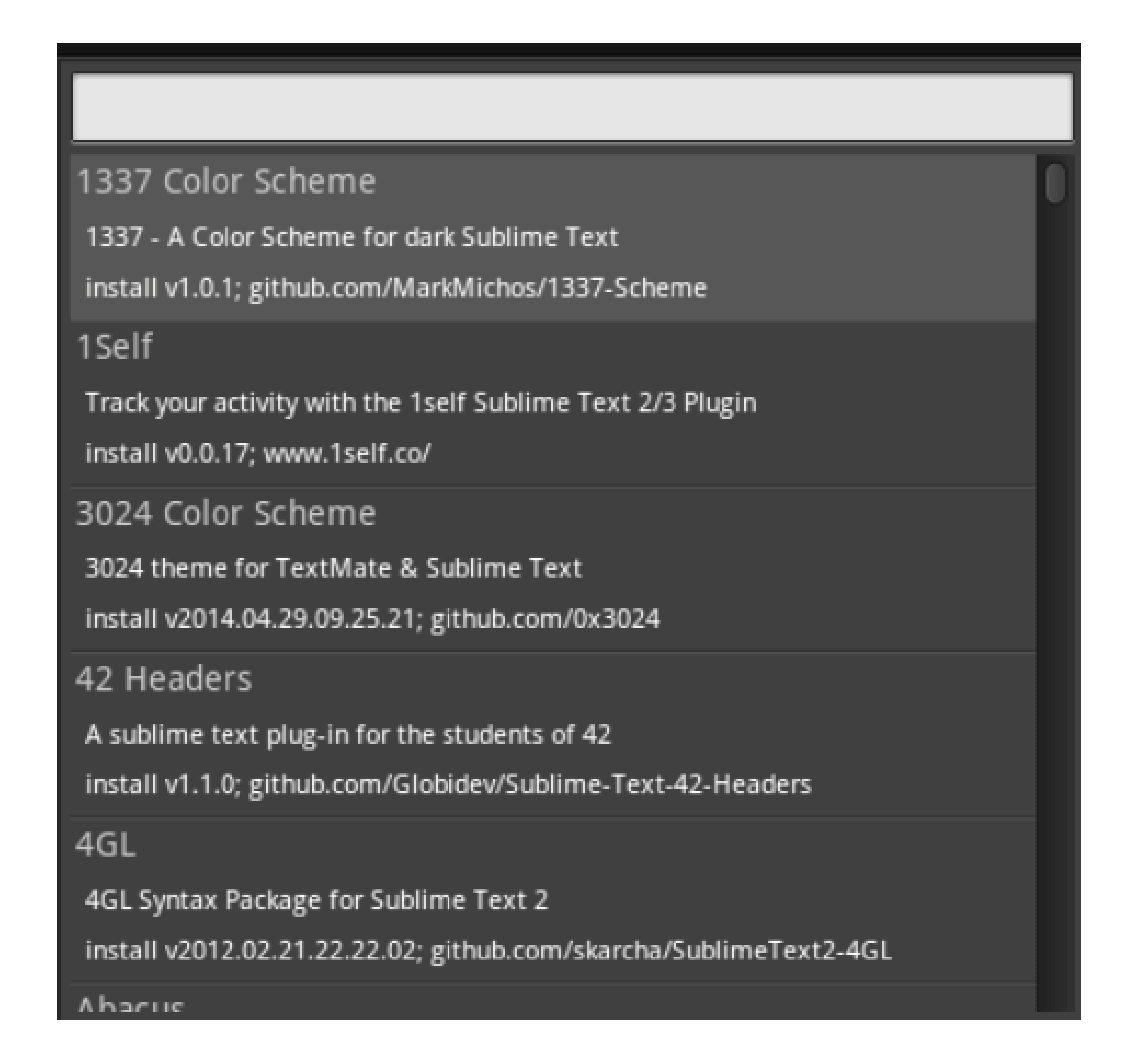


图 1-6　Python 插件的安装

在其中输入“Anaconda”(这是 Sublime Text 3 中最好的一个 Python 插件)，然后按 Enter 键，片刻后即出现如图 1-7 所示的界面，表明 Anaconda 已安装完毕。

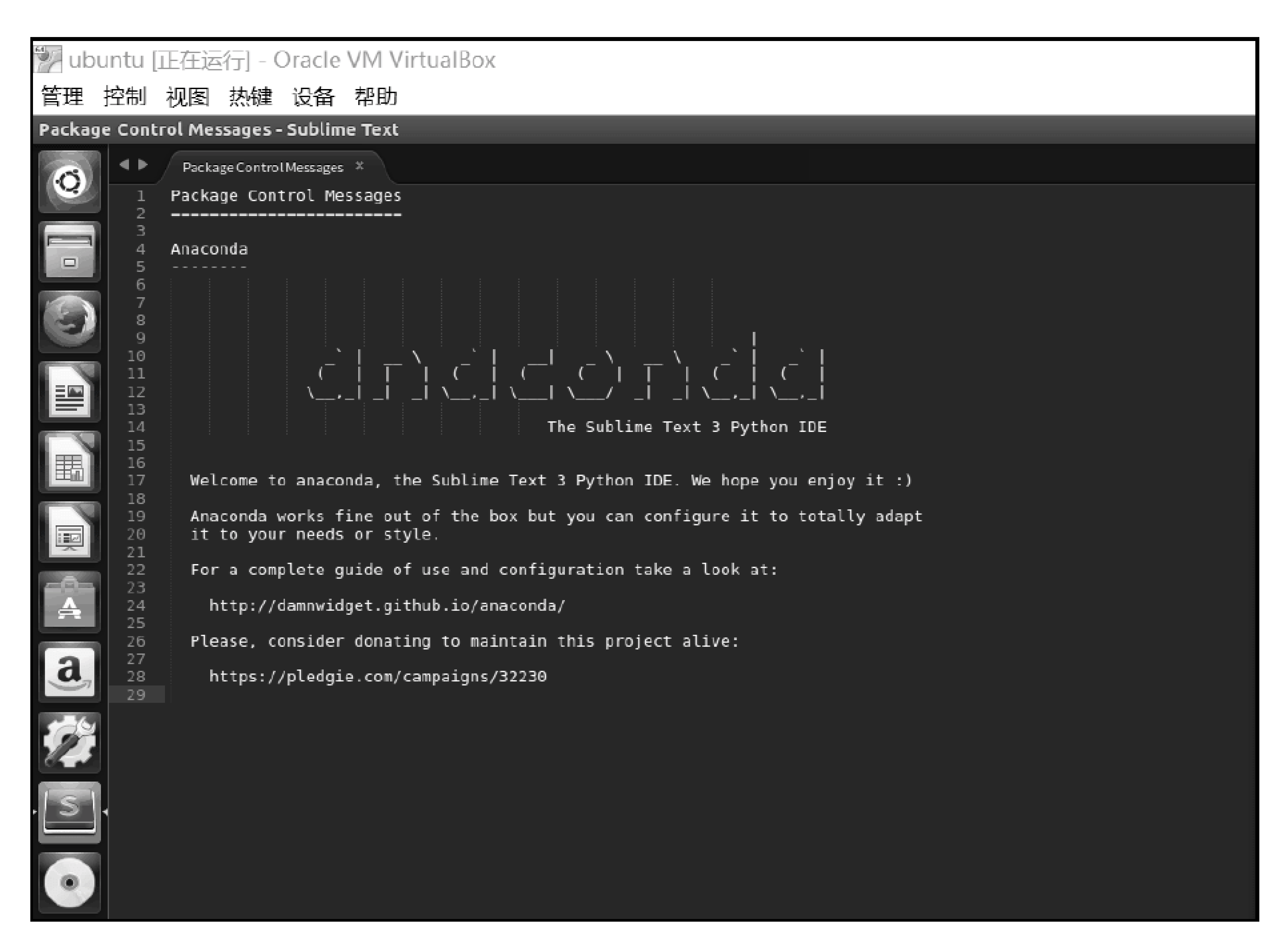


图 1-7　Python 插件的安装

1.2.3　使用 IDLE 来帮助学习 Python

1. IDLE 的安装

如前所述，安装 Python 3 时，默认会得到一个 IDLE(图 1-4)，这是 Python 的集成开发环境，尽管简单，但极为实用。

2. IDLE 的启动

前一小节已介绍了 IDLE 的启动方法。IDLE 启动后，窗口标题栏显示的是 Python 3.x.x Shell，窗口内显示提示符“>>>”，光标紧随其后，表明已经准备就绪，可以在光标处输入代码了。Shell 获取代码语句后会立即执行，并在屏幕上显示结果。

例如，我们在>>>后键入语句：

```
print('This is a command.')
```

则屏幕显示：

```
This is a command.
```

接着，我们计算两个变量相加的值，输入：

```
a=123
b=456
print (a+b)
```

屏幕立即显示计算结果 579。

上述过程如图 1-8 所示。

```
Python 3.4.3 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print('This is a command.')
This is a command.
>>> a=123
>>> b=456
>>> print(a+b)
579
>>> |
Ln: 9 Col: 4
```

图 1-8　直接在 Python 集成编辑环境 IDLE 下输入命令

上面的例子表明，我们可以通过 Shell 在 IDLE 内部执行 Python 命令。除此之外，IDLE 还自带编辑器、解释器和调试器，其中，编辑器用来编辑 Python 程序(或者脚本)，解释器用来解释执行 Python 语句，调试器用来调试 Python 脚本。下面我们从 IDLE 的编辑器开始介绍。

3. 利用 IDLE 创建 Python 程序

IDLE 为开发人员提供了很多有用的特性，如自动缩进、语法着色显示、代码自动完成以及缓存命令历史等，这些功能可以帮助我们高效地编写程序。我们通过一个示例程序对这些特性分别加以介绍，示例程序的源代码如下：

```
#提示用户输入数据
integer1 = input('请输入第一个整数:')
integer1 = int(integer1)
integer2 = input('请输入第二个整数:')
integer2 = int(integer2)
if integer1>integer2:
    print ('%d > %d' %(integer1,integer2))
else:
    print ('%d <= %d' %(integer1,integer2))
```

下面演示如何利用 IDLE 的编辑器来创建 Python 程序。

要新建一个文件，首先从 File 菜单中选择 New File 菜单命令，这样就可以在出现的窗口中输入源程序了。

在输入源程序的过程中，我们会体验到 IDLE 的几个非常方便的功能。

(1)　自动缩进功能。

这里介绍一下 IDLE 的自动缩进功能。事实上，很少有哪种语言能够像 Python 这样重视缩进了，在其他语言(比如 C)中，缩进对于代码的编写来说是“有了更好”，而不是“没有不行”，它充其量是个人书写代码的风格问题；但是到了 Python 这里，则把缩进提升到了语法的高度。换言之，复合语句不是用大括号{}之类的符号表示，而是通过代码缩进来表示。这样做的好处，就是减少了程序员的自由度，有利于统一风格，使得人们在阅读代码时更加轻松。为此，IDLE 提供了自动缩进功能，它能将光标定位到下一行的指定位置。当我们键入与控制结构对应的关键字，如 if 等，或者输入诸如 def 等与函数定义对应的关键字时，按下 Enter 键后，IDLE 就会启动自动缩进功能。

如图 1-9 所示，当 if 语句连同其后的冒号输入完毕并按 Enter 键之后，IDLE 将自动进行缩进。一般情况下，IDLE 将代码缩进一级，即 4 个空格。如果想改变这个默认的缩进量，可以从 Format 菜单中选择 New indent width 命令进行修改。

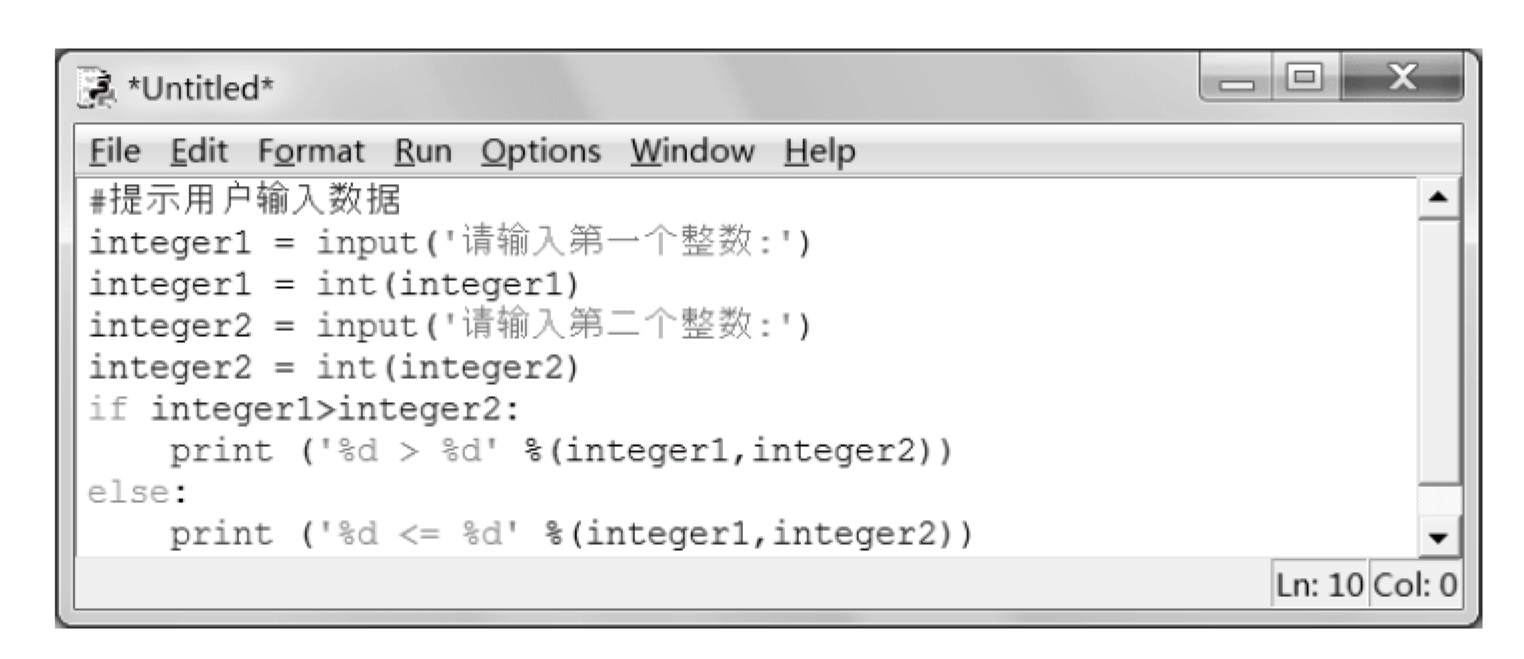


图 1-9　Python IDLE 的自动缩进功能

(2)　语法着色显示。

这里介绍一下 IDLE 的语法着色显示。所谓语法着色显示，就是使用不同的颜色来突出显示代码中的不同元素，关于这一点，我们从图 1-8 和图 1-9 所示的界面中能够看到。默认情况下，关键字显示为橙色，内建函数显示为紫色，字符串显示为绿色，输出的所有结果均显示为蓝色。在键入代码时，会自动应用这些颜色突出显示。如果想改变这些颜色，可以调整 IDLE 的首选项。

语法着色显示的好处在于，可以更容易地区分不同的语法元素，从而提高了程序的可读性；与此同时，语法着色显示还降低了出错的可能性。例如，如果输入的变量名显示为橙色，那就说明该变量名与 Python 的关键字冲突，此时必须给变量重新命名。

(3)　代码自动完成。

这里介绍一下代码自动完成功能。代码自动完成指的是，当用户输入单词的前几个字母后按 Tab 键时，IDLE 会在当前光标处弹出 10 个选项，用户可以从中选择所需要的单词或单词组合(函数名、变量名等)，如果找不到，还可以用上下箭头键进行查找。例如，输入 pr 后按 Tab 键，弹出 10 个提示选项，print 处于首位，按空格键即可将其输入。

程序创建好之后，从 File 菜单中选择 Save 命令保存该程序。若是新文件，则系统将弹出 Save as 对话框，提示输入文件名和保存位置。保存之前，窗口标题栏的名称是“*Untitled*”；保存之后，指定的文件名会自动替代“*Untitled*”。如果文件改动后尚未存盘，则标题栏的文件名前后会出现星号，旨在提醒操作人员。

4.　IDLE 常用的编辑功能

现在我们介绍一下编辑 Python 程序时常用的 IDLE 选项。我们按照不同的菜单分别列出，供初学者参考。

(1)　对于 Edit 菜单，除了上面介绍的几个之外，常用的选项如下。

Undo：撤销上一次的修改。

Redo：重复上一次的修改。

Cut：将所选文本剪切至剪贴板。

Copy：将所选文本复制到剪贴板。

Paste：将剪贴板的文本粘贴到光标所在位置。

Find：在窗口中查找单词或模式。

Find in files：在指定的文件中查找单词或模式。

Replace：替换单词或模式。

Go to line：将光标定位到指定行首部。

(2) 对于 Format 菜单，常用的选项如下。

Indent region：使所选内容右移一级，即增加缩进量。

Dedent region：使所选内容左移一级，即减少缩进量。

Comment out region：将所选内容变成注释。

Uncomment region：去除所选内容每行前面的注释符。

New indent width：重新设定制表位缩进宽度，范围为 2~16，当宽度为 2 时，相当于 1 个空格。

Expand word：代码自动完成。

Toggle tabs：打开或关闭制表位。

5. 在 IDLE 中运行 Python 程序

欲使用 IDLE 执行程序，可从 Run 菜单中选择 Run Module 命令，该命令的功能是执行当前文件。对于前面的示例程序，执行情况如图 1-10 所示。

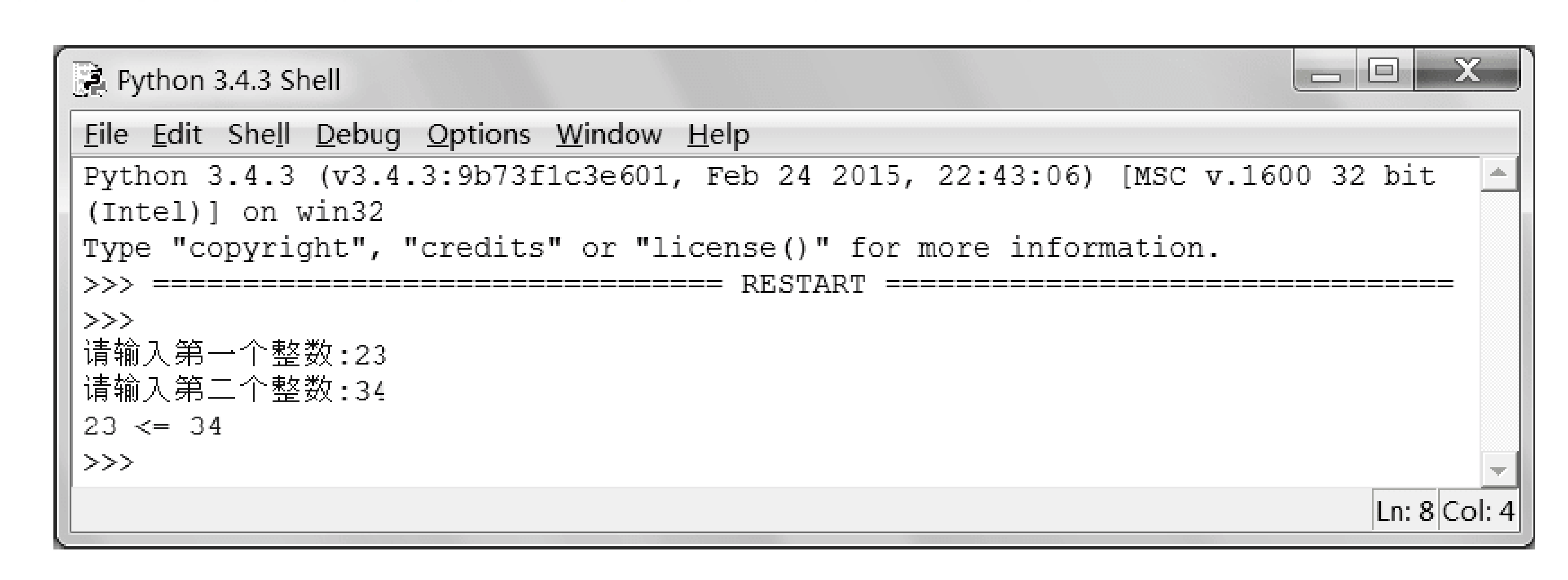


图 1-10　示例程序的运行结果

6. IDLE 调试器的使用

软件开发过程中出错是在所难免的，有语法方面的错误，也有逻辑方面的错误。对于语法错误，Python 解释器能轻而易举地检测出来，此时它会停止程序运行，并显示出错信息；而对于逻辑错误，解释器就鞭长莫及了，此时程序会一直往下执行，但得到的结果却是不正确的。有鉴于此，我们常常需要调试程序。

最简单的调试方法莫过于直接显示程序的中间数据。例如，可以在程序的某些关键位置插入 print 语句，用以显示某些中间变量的值，从而判断出错与否。但这个办法比较麻烦，因为开发人员必须在所有可疑的地方都插入 print 语句，待程序调试完后，还必须删除这些语句，很繁琐。

Python 中自带了一个调试模块 pdb，专门用于调试程序，本书 5.8 节中将详细介绍，本章仅介绍 IDLE 自身提供的调试功能。

IDLE 提供了一个调试器，为我们查找逻辑错误带来了诸多方便。利用调试器，可以

分析被调程序的数据，并监视程序的执行流程。调试器的功能很强，包括暂停程序执行、检查和修改变量、调用方法而不更改程序代码等。下面简要介绍一下 IDLE 调试器的使用方法。

在 Python Shell 窗口中，选择 Debug 菜单中的 Debugger 命令，便可启动 IDLE 的交互式调试器。此时，IDLE 将打开 Debug Control 窗口，并在 Python Shell 窗口中输出[DEBUG ON]，后跟提示符“>>>”。这样一来，我们就可以像平时那样使用这个 Python Shell 窗口了，所不同的是，现在输入的任何命令都必须是在调试器下允许使用的命令。还可以在 Debug Control 窗口中查看局部变量和全局变量等有关内容。要退出调试器，可以再次单击 Debug 菜单中的 Debugger 命令，这时 IDLE 会关闭 Debug Control 窗口，并在 Python Shell 窗口中输出[DEBUG OFF]。

7. IDLE 的命令历史功能

在 DOS/Unix 等以命令行方式操作的操作系统中，用户使用上下箭头键可以方便地调出以前用过的各种命令，颇为方便，这一功能称为“命令历史”功能。命令历史可以记录会话期间在命令行中执行过的所有命令。在 IDLE 提示符下，按 Alt+P 组合键可以找回这些命令。每按一次，IDLE 就会从最近的命令开始检索命令历史，并按命令使用的顺序逐个显示。按 Alt+N 组合键，则可以反向遍历各个命令，即从最初的命令开始遍历。

8. IDLE 小结

IDLE 是 Python 安装包自带的一个集成开发环境，虽相对简单，但对 Python 的初学者非常适用。本小节通过一个示例程序详细介绍了 IDLE 在 Python 编程中的使用方法，希望读者能够熟练使用。另外，除非特殊指明，本书所用的集成开发环境均指 IDLE。

1.2.4 Python 常用的开发工具

这里所说的开发工具，实际上指的就是集成开发环境 IDE。一个优秀的 IDE，最重要的要求就是在完成一般文本编辑的基础上，能够提供针对特定语言的各种快捷编辑功能，让程序员尽可能快捷、舒适、清晰地浏览、输入、修改代码。对于一个现代 IDE 来说，语法着色、错误提示、代码自动完成、代码折叠、代码块定位、重构，及与调试器、版本控制系统(Version Control System，VCS)的集成等，都是重要的功能。以插件、扩展系统为代表的可定制框架，已成为现代 IDE 的另一个流行趋势。

但 IDE 并非功能越多越好，这是因为，更多的功能往往意味着更大的复杂度和更大的开销，这不但会分散程序员的精力，而且还可能带来更多的错误。一般来讲，满足基本功能需要、符合自己使用习惯的 IDE，就是最好的 IDE。程序员的逻辑永远是：用最合适的工具做最合适的事情。

正因如此，比起大而全的 IDE，以单纯的文本编辑器结合独立的调试器、交互式命令行等外部小工具，也是另一种开发方式。由于 Python 本身的简洁性，在书写小的代码片段以及通过示例代码学习时，这种方式尤其适合。

这里简单介绍 Python 程序员中最流行的几款 IDE。

(1) 内置 IDE。Python 的各个常见发行版均有内置的 IDE，虽然它们的功能一般不够强大、完整，但简便易得，对于初学者来说，它们也是上手的最好选择，可以让用户更专注于语言本身，而不会因 IDE 的繁琐和复杂而分散精力。

(2) IDLE。IDLE 是 Python 的官方标准开发环境，简单而小巧，但具备了 Python 应用开发的几乎所有功能，包括了交互式命令行、编辑器、调试器等基本组件，足以满足大多数简单应用的要求。IDLE 是用纯 Python 基于 Tkinter 编写的，最初的作者正是 Python 之父 Guido van Rossum 本人。

(3) PythonWin。PythonWin 是 Python Win32 Extensions(半官方性质的 Python for Win32 增强包)的一部分，也包含在 ActivePython 的 Windows 发行版中，该版本只针对 Win32 平台。

总体来说，PythonWin 是一个增强版的 IDLE，其优势体现在易用性方面(如同 Windows 本身的风格)。除了易用性和稳定性之外，(简单的)代码完成和更强的调试器功能相对于 IDLE 都是明显优势。

(4) MacPython IDE。MacPythonIDE 是 Python 的 Mac OS 发行版内置的 IDE，可视为 PythonWin 的 Mac 对应版本，由 Guido 的兄长 Just van Rossum 编写。

(5) Emacs 和 Vim。Emacs 和 Vim 号称是全球最强大的两个文本编辑器，对于许多程序员来说，它们是万能 IDE 的最佳选择。比起同类的通用文本编辑器(如 UltraEdit)，Emacs 和 Vim 由于扩展功能十分强大，可以有针对性地搭建出更为完整便利的 IDE。

虽然掌握二者之后可谓终身受益，但其学习曲线都比较陡峭。由于历史原因，它们的设计理念都是基于纯 ASCII 字符环境，GUI 相对来说不是支持的重点，只有大量使用快捷键才能带来最大的便利。对于初学者来说，相对而言 Vim 更简洁一些，但 Emacs 的 GUI 与一般编辑器的习惯更接近些。

(6) Eclipse + PyDev。Eclipse 是新一代优秀的泛用型 IDE，虽然它是基于 Java 技术开发的，但出色的架构使其具有不逊于 Emacs 和 Vim 的可扩展性，目前已经成为许多程序员最爱的一把“瑞士军刀”。

PyDev 是 Eclipse 上的 Python 开发插件中最为成熟、最为完善的一个，而且还在持续的活跃开发之中。除了 Eclipse 平台提供的基本功能之外，PyDev 的自动代码完成、语法查错、调试器、重构等功能都做得相当出色，可以说是开源产品中最为强大的一个，许多贴心的小功能也很符合用户的编辑习惯，使用起来得心应手。

但有其利必有其弊，无论是 Eclipse 还是 PyDev，在速度和资源占用方面都是致命的，在低配置机器上运行起来比较吃力。

(7) UliPad。UliPad 是国内知名的 Python 开发工具，是由 PythonCN 社区核心成员 Limodou 开发的 IDE。因为 UliPad 是基于 wxPython 开发的，所以安装 UliPad 之前需要先安装 wxPython。

(8) SPE(Stani’s Python Editor)。SPE 是很有特色的一个轻量级的 Python IDE，功能很全面而不失小巧轻便，特别适合书写小的脚本。

即时生成代码的 UML 类图是其独树一帜的功能。除此之外，SPE 还特别注重与外部工具的集成。例如，集成了 wxGlade 作为所见即所得的 GUI 开发环境，集成了 Winpdb 作为调试器，甚至还能与 3D 建模工具 Blender 集成。

SPE 没有管理 Project 的概念，当开发多文件、多目录组成的项目时会感到不便。此外，界面设计相对来讲不够细致，也算是瑕疵。

(9) WingIDE。WingIDE 是 Wingware 公司开发的商业化产品，总体来说，是目前最为强大、最为专业的 Python IDE。其最大的缺点和 PyDev 一样，就是资源占用比较多，导致速度慢。

(10) 其他 IDE。除了上述几款 IDE 之外，还有很多常用的 IDE，此处不再一一列举，比较著名的有 Pycharm、Eric3、Komodo、Textmate、Scribes、Intype、PyScripter 等。

1.2.5 “Hello World!”——第一个 Python 程序

在 1.2.1 小节中介绍 Python 开发环境的搭建时，我们演示了在 DOS 命令提示符窗口中直接输出字符串“Hello World!”的例子。尽管单一的语句能够在屏幕上输出所需的内容，但一般来讲，用户可能更需要编写 Python 程序来实现特定的业务逻辑，同时也方便代码的不断完善和重复利用，毕竟直接使用交互模式不是很方便。此时可以使用 1.2.3 小节中介绍的方法创建程序文件，输入程序并保存为文件(务必保证扩展名为 py)后，执行 Run→Run Module 命令运行，程序运行结果将直接显示在 IDLE 交互界面上。除此之外，也可以在资源管理器中双击扩展名为 py 的 Python 程序文件来运行；在某些情况下，还可能需要在 DOS 命令提示符环境下运行 Python 程序文件。在“开始”菜单的“附件”中单击“命令提示符”，然后执行 Python 程序。例如，假设有程序 hw.py，其内容如下：

```
def main():
    print('Hello World!')
main()
```

在 IDLE 环境中运行该程序后，结果如图 1-11 所示。在 DOS 命令提示符环境下运行该程序后，结果如图 1-12 所示。这里展示了运行 Python 程序的两种方法，虽然第二种方法看上去更加简单，但是我们推荐使用第一种方法来运行 Python 程序。

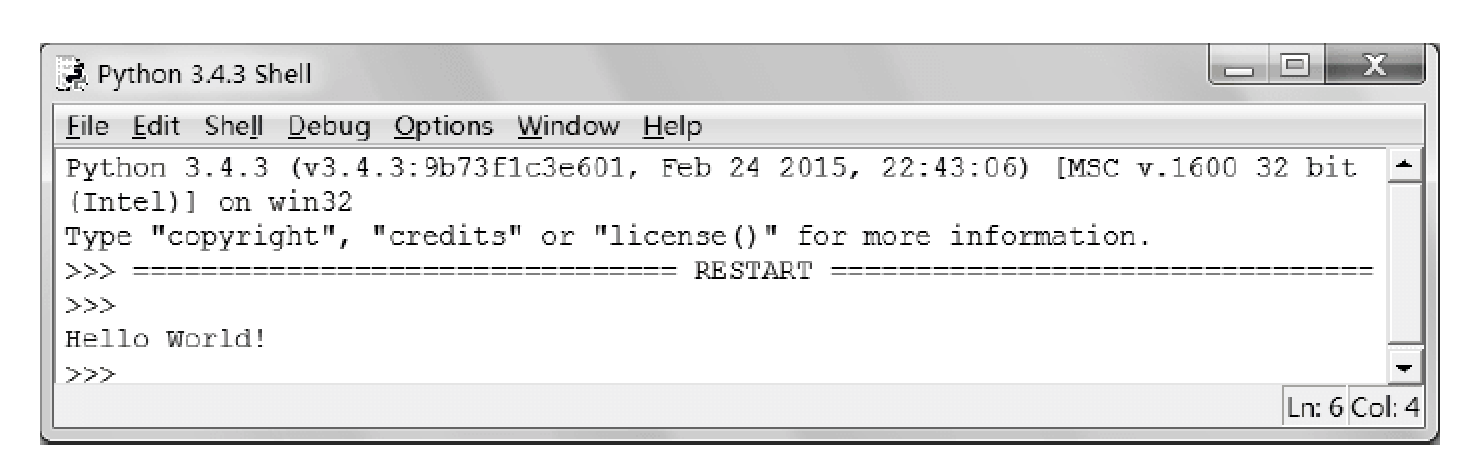


图 1-11　在 IDLE 中运行程序

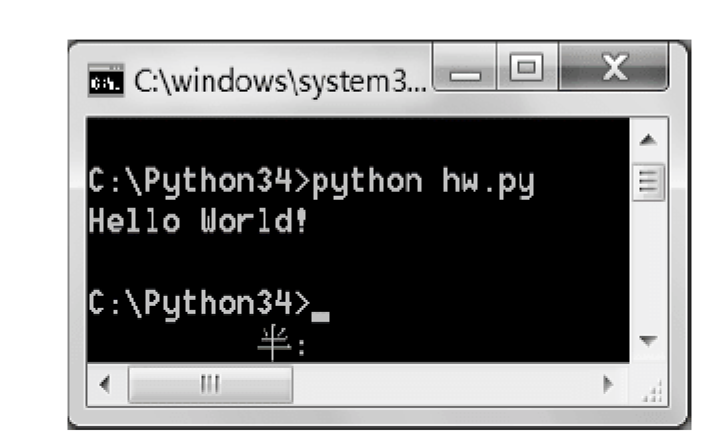


图 1-12　在命令提示符下运行程序

本 章 小 结

Python 是一种代表简单主义思想的语言，但是它功能强大，应用广泛。我们在使用 Python 编程时，经常会感觉到它的方便、快捷。在本章中，我们对 Python 进行了简单的介绍，了解了什么是 Python，为什么要学习 Python；简单讲述了 Python 的发展历史和流行版本，详解了搭建 Python 环境的基本方法，重点介绍了 IDLE 开发工具的使用方法，以及

用其学习 Python 的优势所在。除了 IDLE 之外，本章还简单地介绍了数种常用的 Python 开发工具，供读者自行选用。

习　　题

1. 填空题

(1) Python 是一种(　　　)语言，写好代码即可直接运行，省去了(　　　)、(　　　)等一系列麻烦，这对于需要多动手实践的初学者而言，减少了很多出错的机会。

(2) Python 支持(　　　)的操作方式，如果只是运行一段简单的小程序，连编辑器都可以省略，直接输入即可运行。

(3) Python 是一种结构清晰的编程语言，使用(　　　)的方式来表示程序的嵌套关系，将过去软性的编程风格升级为硬性的语法规定。

(4) Python 3.x 将 print 视为一个(　　　)而不是命令。

(5) 欲使用 IDLE 执行程序，可从 Run 菜单中选择(　　　)命令，该命令的功能是执行当前文件。

2. 选择题

(1) 哪一个不是 Python 的特点？(　　　)

A. 简洁性　　B. 可扩展性　　C. 解释性　　D. 万能性

(2) 第一个 Python 编译器是哪一年问世的？(　　　)

A. 1989 年　　B. 1991 年　　C. 1999 年　　D. 2001 年

(3) 哪一个不是 Python 语言的基本结构？(　　　)

A. 链表　　B. 元组　　C. 字典　　D. 集合

(4) Python 从哪个版本开始全面支持 Unicode？(　　　)

A. 1.0　　B. 2.0　　C. 2.7　　D. 3.0

(5) 下面哪个不是 Python 的与众不同特性？(　　　)

A. 易用性与高速度的完美结合　　B. 成熟的跨平台技术

C. 具有丰富和强大的基本类库　　D. 简易的 CORBA 绑定

3. 问答题

(1) 什么是 Python？请简单叙述一下。

(2) 在输入源程序的过程中，IDLE 有哪几个很方便的功能？

(3) Python 常用的开发工具有哪些？



第 2 章

Python 语言基础

本章要点

(1) Python 的语法和句法。
(2) Python 的数据类型。
(3) Python 的常量与变量。
(4) Python 的运算符与优先级。
(5) Python 的数值类型。
(6) Python 的字符串类型。
(7) Python 的高级数据类型(列表、元组、字典、集合)。
(8) 正则表达式。

学习目标

(1) 了解 Python 的语法和句法。
(2) 理解 Python 的数据类型。
(3) 掌握 Python 的常量与变量。
(4) 掌握 Python 的数值、字符串、列表、元组、字典、集合。
(5) 掌握正则表达式的概念及其应用。

本章将以较大的篇幅介绍 Python 语言最基础的内容，包括 Python 语法、基本数据类型、常量变量、运算符与优先级、高级数据类型，最后介绍正则表达式的概念及其应用。

2.1 基础 Python 语法

2.1.1 标识符

在编程语言中，标识符就是程序员自己规定的具有特定含义的词，比如类名称、属性名、变量名、函数名等。一般语言规定，标识符由字母或下划线开头，后面可以跟字母、数字、下划线。

Python 标识符的命名规则如下。

(1) 标识符长度无限制。

(2) 标识符不能与关键字(见附录 A)同名。

(3) 字母大小写敏感。

(4) 在 2.x 版本的 Python 中，标识符的命名规则与一般语言的规定一样，但在 3.x 的 Python 中进行了扩展，标识符的引导字符可以是字母、下划线以及大多数非英文语言的字母，只要是 Unicode 编码的字母均可，后续字符可以是上述任意字符，也可以是数字。

虽然 Python 对标识符命名的限制很少，但使用时，仍需要注意以下约定。

(1) 不要使用 Python 预定义的某些标识符，因此要避免使用诸如 NotImplemented 等名字，这些在未来有可能被 Python 的新版本使用。

(2) 不要使用 Python 内建函数名或内置数据类型或异常名作为标识符的名字。

(3) 不要在名字的开头和结尾都使用下划线，因为 Python 中大量地采用这种名字定义各种特殊的方法和变量。

2.1.2　Python 语法和句法

Python 语句中有一些基本规则和特殊字符。

(1)　井号(#)表示其后的字符为 Python 语句的注释。

(2)　换行(\n)是标准的行分隔符(通常一个语句占一行)。

(3)　反斜线(\)继续上一行。

(4)　分号(;)将两条语句放在一行中。

(5)　冒号(:)将复合语句的头和体分开。

(6)　代码块(语句块)用缩进的方式体现。

(7)　用不同的缩进深度分隔不同的代码块。

(8)　Python 文件以模块的形式组织。

1. 注释(#)

尽管 Python 是可读性最好的语言之一，但这并不意味着代码中的注释可以不要。Python 的注释语句以#字符开始，注释可以在一行的任何地方开始，解释器将会忽略该行#之后的所有内容。

2. 续行(\)

一般来讲，Python 的相邻语句使用换行(回车)分隔，亦即一行一条语句。如果一行语句过长，可以使用续行符(\)分解为多行，例如：

```
print("This line is tooooooooooo \
long")
```

关于续行符有两种例外情况。

(1)　一个语句不使用反斜线也可以跨行书写。

在使用闭合操作符时，单一语句也可以跨多行。例如，在含有小括号、中括号、花括号时，可以多行书写：

```
print("This is a multiline",
     "example")
```

但须注意，这时的缩进(即使是自动的缩进)将失去语法上的作用。

(2)　三引号内包含的字符串也可以跨行书写。例如：

```
print('''hi there, this is a long message for you
that goes over multiple lines!''')
```

如果要在使用反斜线换行和使用括号元素换行之间做一个选择的话，我们推荐使用后者，因为这样可读性会更好。

3. 多个语句构成代码组(:)

缩进位置相同的一组语句形成一个语句块，亦称代码块或代码组。像 if、for、while、def 和 class 之类的复合语句，首行均以关键字开始，并以冒号(:)结束，该行之后的一行或多行代码就构成了代码组，即语句块。

4. 代码组以不同的缩进分隔

Python 使用缩进来分隔代码组。代码的层次关系是通过相同深度的空格或制表符缩进来体现的，同一代码组内的代码行左边必须严格对齐。换言之，一个代码组内的各行代码，左边必须有数目相同的空格或数目相同的制表符，而且不能以一个制表符替代多个空格！如果不严格遵守这一规则，同一组的代码就可能被视为另一个组，轻则导致逻辑错误，重则导致语法错误。

注意： 对初次使用空白字符作为代码块分界的人来说，首先遇到的问题是：缩进几个空格或制表符才算合适？理论上讲是没有限制的，但我们推荐使用 4 个空格。需要说明一点，不同的文本编辑器中制表符代表的空白宽度不一样，如果所写的代码要跨平台应用，或者将来要被不同的编辑器来读写，那么建议不要使用制表符。

随着缩进深度的增加，代码块的层次也在逐步加深，未缩进的代码块处于最高层次，称作脚本的 main 部分。

采用缩进对齐方式来组织代码，不但代码风格优雅，而且其可读性也大大增强。不仅如此，这种方法还有效地避免了“悬挂 else”(dangling-else)问题，同时也避免了未写大括号时的单一子句问题。试想，如果 C 语言的 if 语句后漏写大括号，而后面却跟着两个缩进的语句，结果会如何呢？毫无疑问，无论条件表达式是否成立，第二个语句总会被执行。这种问题很难调试，不知困惑了多少程序员。

5. 同一行书写多个语句(;)

Python 允许将多个语句写在同一行上，语句之间用分号隔开，而这些语句也不能在这行开始一个新的代码块。例如：

```
a=10; b=20; print(a+b)
```

但必须指出，同一行上书写多个语句，会使代码的可读性大大降低。Python 虽然允许这样做，但并不提倡这么做。

6. 模块

每个 Python 脚本文件均可视为一个模块，它以磁盘文件的形式存在。如果一个模块规模过大，包含的功能太多，就应该考虑对该模块进行拆分，即拆出一些代码另外组建一个或多个模块。模块里的代码既可以是一段直接执行的脚本，也可以是一堆类似库函数的代码，从而可以被别的模块导入(import)后调用。模块可以包含直接运行的代码块、类定义、函数定义，或它们的组合。

2.2 数　　值

2.2.1 数据类型

与 C 等编译型语言不同，Python 中的变量无须声明。每个变量在使用前都必须赋值，

变量赋值以后，该变量才会被创建。

在 Python 中，变量就是变量，它没有类型，我们所说的“类型”是变量所指的内存中对象的类型。

赋值运算符(=)用来给变量赋值。

赋值运算符左边必须是一个变量名，右边是存储在变量中的值。例如：

```
>>> counter = 100     # 整型变量
>>> miles = 1000.0    # 浮点型变量
>>> name = "python"   # 字符串
>>> print (counter); print (miles); print (name)
100
1000.0
python
>>> |
```

Python 允许同时为多个变量赋值，例如：

a = b = c = 1

注意：

- 以上赋值语句创建了一个整型对象，值为 1，三个变量被分配到相同的内存空间上。以下的运行结果可以证实这一点：

```
>>> a=b=c=1
>>> print("a的地址:",id(a))
a的地址: 1693176272
>>> print("b的地址:",id(b))
b的地址: 1693176272
>>> print("c的地址:",id(c))
c的地址: 1693176272
>>> |
```

- Python 中的一切事物皆为对象，并且规定参数的传递都是对象的引用，因此，对象的赋值实际上是对象的引用。

也可以为多个对象指定多个变量，例如：

```
a, b, c = 1, 2, "python"
```

以上形式的赋值语句也称作复合赋值语句，两个整型对象 1 和 2 分配给变量 a 和 b，字符串对象“python”分配给变量 c。

Python 3 中有 6 种标准的数据类型，它们是 Number(数值)、String(字符串)、List(列表)、Tuple(元组)、Dictionary(字典)、Sets(集合)。本节介绍数值类型，后续各节分别介绍其余的各种类型。

Python 3 支持 4 种数值类型：int(整型)、float(浮点型)、bool(布尔型)、complex(复数型)。

在 Python 3 里，只有一种整数类型 int，表示为长整型，而不像 Python 2 那样区分标准整型与长整型。

与大多数编程语言一样，数值类型的赋值和计算都是很直观的。内建的函数 type()可以用来查询变量所指的对象类型。例如：

```
>>> a, b, c, d = 10, 3.5, True, 2+4j
>>> print(type(a), type(b), type(c), type(d))
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
>>>
```

注意： 在 Python 2 中没有布尔型，它像 C 语言那样，用数字 0 表示 False，用 1 表示 True。在 Python 3 中，把 True 和 False 定义成关键字了，但它们的值仍然是 1 和 0，它们可以和数字参与运算。

当指定一个值时，就创建了一个 Number 对象：

```
var1 = 1
var2 = 2
```

同样，这里的 var1 和 var2 是两个对象引用。使用 del 语句可以删除对象引用。del 语句的语法是：

```
del var1[,var2[,var3[...,varN]]]
```

例如：

```
del var
del var_a, var_b
```

一个变量可以通过赋值指向不同类型的对象。例如：

```
>>> a=10
>>> print(type(a))
<class 'int'>
>>> a="10"
>>> print(type(a))
<class 'str'>
>>>
```

1. 数值运算

像其他语言一样，Python 的数值运算包括加、减、乘、除四则运算以及取余、乘方运算等。例如：

```
>>> 5 + 4              # 加法
9
>>> 4.3 - 2            # 减法
2.3
>>> 3 * 7              # 乘法
21
>>> 2 / 4              # 除法，得到一个浮点数
0.5
>>> 2 // 4             # 除法，得到一个整数
0
>>> 17 % 3             # 取余
2
>>> 2 ** 5             # 乘方
32
>>>
```

注意：

- 除法运算符“/”总是返回一个浮点数，要返回整数应使用“//”运算符。
- 在混合运算时，Python 把整型数转换成为浮点数。
- 除上述运算外，Python 的整数还支持“位运算”，详见表 2-1。

2. 数值类型实例

Python 3 支持的数值类型实例如下。

int 型实例：10、100、-786、080、-0490、-0x260、0x69。

float 型实例：0.0、15.20、-21.9、32.3e+18、-90.、-32.54e100、70.2e-12。

bool 型实例：True、False。

Python 还支持复数。复数由实部和虚部构成，可以表示为 a + bj 或者 complex(a,b)，其中实部 a 和虚部 b 都是浮点型。

complex 型实例：3.14j、45.j、9.322e-36j、.876j、-.6545+0j、3e+26j、4.53e-7j。

2.2.2　常量与变量

几乎所有的编程语言都有变量和常量的概念，它们与数学上的概念很相似。

1. 变量

变量与数学函数中的变量一样。顾名思义，变量就是其值可以改变的量，只不过在程序中变量不仅仅可以是数字，还可以是字符串等。变量名的命名方式严格遵循上一节提及的 Python 标识符命名规则，即由英文字母、Unicode 字符(含汉字)、数字、下划线组成，且不能以数字开头。

下面的例子通过赋值运算符把 1 赋值给 a，这个 a 就是我们创建的一个变量：

```
>>> a=1
>>> a
1
>>> |
```

又如：

```
a=1
b=2
```

这里我们用一个字母来表示变量，通过赋值运算符来给变量赋值。

Python 是动态语言，所以我们不需要像 C 语言那样提前声明一个变量的数据类型。C 语言使用变量之前需要声明一下。例如：

```
int a=1;
```

而 Python 直接使用 a=1 即可。这就是动态语言的好处，我们无须关心变量本身的数据类型，Python 有自己的判断机制。

注意：给变量赋值的时候，Python 只会记住最后一次所赋的值。例如：

```
>>> a=1
>>> a=2
>>> a
2
>>> 
```

先给 a 赋值 1，再赋值 2，最后输出 a 的时候显示的是最后一次所赋的值 2。

2. 常量

常量与变量相对应，就是程序运行过程中其值不可改变的量。例如：

```
PI=3.1415926
```

这时，我们就认定 PI 是个常量，也就是说，PI 始终代表 3.1415926。实际上，Python 中并没有严格意义的常量，因为 Python 中没有保证常量不会改变的机制。一般来讲，我们使用全部大写的字母来表示常量。尽管我们称 PI 为常量，但仍然可以给 PI 重新赋值。

换言之，Python 的世界里本来就没有常量，编程时主动不修改的变量也就伪装成了常量。用大写字母来“注明”常量，只有提醒的意义，其实这个值还是可以改变的。

在编程过程中，我们可能会用到常量的概念，所以这里还是要提一下。

2.2.3 运算符与优先级

1. Python 的运算符

Python 的运算符十分丰富，表 2-1 简要列示了 Python 的运算符及其用法。

表 2-1 Python 运算符及其用法

运算符	名　称	说　明	例　子
+	加	两个对象相加(对字符串则是连接)	3 + 5 得到 8。 ‘a’ + ‘b’ 得到 ‘ab’
-	减/负号	得到一个负数，或是一个数减去另一个数	-5.2 得到一个负数。 50 - 24 得到 26
*	乘	两个数相乘或是返回一个被重复若干次的字符串	2*3 得到 6。 ‘la’*3 得到 ‘lalala’
**	幂	指数运算，返回 x 的 y 次幂	3**4 得到 81 (即 3*3*3*3)
/	除	x 除以 y	4/3 得到 1.3333333333333333
//	取商数	返回商的整数部分	4 // 3.0 得到 1.0
%	取余数	返回除法的余数	8%3 得到 2。 -25.5%2.25 得到 1.5
<<	左移	把一个数的位向左移一定数目(每个数在内存中都表示为位或二进制数字，即 0 和 1)	2 << 1 得到 4。因为 2 用二进制表示时为 10B，往左移一位变成二进制的 100B，二进制的 100B 用十进制表示则为 4

续表

运算符	名　称	说　明	例　子
>>	右移	把一个数的位向右移一定数目	11>>1 得到 5。因为 11 用二进制表示时为 1011B，向右移动 1 位后得到 101B，即十进制的 5
&	位和(Bitwise AND)	数的位和	5 & 3 得到 1。(101B &11B 得到 1B)
|	位或(Bitwise OR)	数的位或	5 | 3 得到 7。(101B | 11B 得到 111B)
~	位翻转	位翻转	x 的位翻转是-(x+1)，~5(101B)得到-6(-110B)
<	小于	比较运算符，返回 x 是否小于 y 的结果。返回 1 则表示真，返回 0 则表示假。Python 中也可以用特殊的变量 True 表示真，用 False 表示假。注意首字母为大写	5<3 返回 0(即 False)；而 3<5 返回 1(即 True)。也可以任意连接比较运算符：3 < 5 < 7 返回 True
>	大于	返回 x 是否大于 y	5>3 返回 True。如果两个操作数都是数值，它们会先被转换成相同的类型后才开始计算。如果 x 和 y 类型不同，则会返回 False
<=	小于或等于	返回 x 是否小于或等于 y	x = 3; y = 6; x <= y 返回 True
>=	大于或等于	返回 x 是否大于或等于 y	x = 4; y = 3; x >= y 返回 True
==	等于	比较两个对象是否相等	x = 2; y = 2; x == y 返回 True。x = ‘abc’; y = ‘Abc’; x == y 返回 False
!=	不等于	比较两个对象是否不相等	x = 2; y = 3; x != y 返回 True
not	布尔非	如果 x 为 None(空)、0、False 或空字符串，则 not x 会返回 True，否则 not x 会返回 False	x = True; not x 返回 False。x = False; not x 返回 True
and	布尔与	如果 x 为 False，x and y 返回 False，否则它会返回 y 的计算值	x = False; y = True; x and y 的结果会返回 False。在这个例子里，Python 语言并不会计算 y 的值，因为 x 的值已经是 False，所以这个表达式的值肯定是 False。这个现象称为短路计算
or	布尔或	如果 x 为 True，则返回 True，否则返回 y 的计算值	x = True; y = False; x or y 的结果会返回 True。短路计算在这里也同样适用

2. Python 运算符的优先级

表 2-2 给出 Python 运算符的优先级，其排列顺序是从最低的优先级(最松散地结合)到最高的优先级(最紧密地结合)。这意味着在计算一个表达式时，Python 首先计算列在表下部的运算符，然后再计算列在表上部的运算符。

表 2-2 Python 运算符的优先级

运 算 符	描 述
lambda	Lambda 表达式
or	布尔或
and	布尔与
not x	布尔非
in、not in	成员测试
is、is not	同一性测试
<、<=、>、>=、!=、==	比较
|	按位或
^	按位异或
&	按位与
<<、>>	移位
+、-	加法与减法
*、/、%	乘法、除法、取余数
+x、-x	正负号
~x	按位翻转
**	指数
x.attribute	属性参考
x[index]	下标
x[index:index]	寻址段
f(arguments...)	函数调用
(expression,...)	绑定或元组显示
[expression,...]	列表显示
{key:datum,...}	字典显示
‘expression,...’	字符串转换

上述运算符优先级表决定了哪个运算符在别的运算符之前计算。然而，如果想要改变它们的计算顺序，可以使用圆括号。例如，想要在一个表达式中让加法在乘法之前计算，那么就要写成类似于(1 + 2) * 3 的形式。

3. Python 运算符的结合规律

运算符通常自左向右结合，即具有相同优先级的运算符按照从左向右的顺序计算。例

如，1 + 2 + 3 被解释为(1 + 2) + 3。也有一些运算符是自右向左结合的，如赋值运算符。这样的运算符，其结合顺序正相反，即 a = b = c 被解释为 a = (b = c)。

注意： 合理使用括号能够增强代码的可读性。在很多场合下，使用括号都是一个好主意，而没用括号的话，可能会使程序得到错误的结果，或使代码的可读性降低，引起阅读者的困惑。括号在 Python 语言中不是必须存在的，不过为了获得良好的可读性，使用括号总是值得的。

2.3　字　符　串

1. Python 字符串

字符串(string)是 Python 中最常用的数据类型，Python 使用引号(单引号 ' 或双引号 ")作为字符串的定界符(一般为单行字符串，多行字符串使用三引号，稍后介绍)。

要创建一个字符串，只要用成对的引号把若干个字符括起来即可，例如：

```
var1 = 'Hello World!'
```

```
var2 = "Python Runoob"
```

Python 规定，单引号内可以使用双引号，这时双引号被视为一个普通字符，不再作为定界符，反之亦然。例如：

```
str1='I want a book,\n and you want a book,too.'   #单引号
str2='"I want a book,\n and you want a book,too."'  #单引号中使用双引号
str3="I want a book,\n and you want a book,too."   #双引号
str4="'I want a book,\n and you want a book,too.'"  #双引号中使用单引号
print(str1); print(str2); print(str3); print(str4)
```

```
I want a book,
 and you want a book,too.
"I want a book,
 and you want a book,too."
I want a book,
 and you want a book,too.
'I want a book,
 and you want a book,too.'
>>> 
```

一个字符串用什么引号开头，就必须用什么引号结尾，首尾定界符不匹配时将出错，例如：

```
>>> str="Wrong string'
SyntaxError: EOL while scanning string literal
>>> 
```

2. 字符串中值的访问(子串截取)

与 C 语言不同，Python 不支持字符类型，即使是单个字符，Python 也将其视为一个字符串来使用。

访问子串，实际上就是截取字符串中的子串，有的文献称其为“切片运算”，使用的运算符是方括号([]或[:]或[::])，例如：

```
str="0123456789"
print ("str[0:3]:",str[0:3]) #截取第一位到第三位的字符
print ("str[:]:",str[:]) #截取字符串的全部字符
print ("str[6:]:",str[6:]) #截取第七个字符到结尾
print ("str[:-3]:",str[:-3]) #截取从头开始到倒数第三个字符之前
print ("str[2]:",str[2]) #截取第三个字符
print ("str[-1]:",str[-1]) #截取倒数第一个字符
print ("str[::-1]:",str[::-1]) #创建一个与原字符串顺序相反的字符串
print ("str[-3:-1]:",str[-3:-1]) #截取倒数第三位与倒数第一位之前的字符
print ("str[-3:]: ",str[-3:]) #截取倒数第三位到结尾
print ("str[:-5:-3]:",str[:-5:-3]) #逆序截取
```

上述程序段的运行结果如下：

```
str[0:3]: 012
str[:]: 0123456789
str[6:]: 6789
str[:-3]: 0123456
str[2]: 2
str[-1]: 9
str[::-1]: 9876543210
str[-3:-1]: 78
str[-3:] :  789
str[:-5:-3]: 96
>>> |
```

从上面的例子可以看出：①要获得单个字符，应使用切片运算符 s[i]，但 i 不能省略；②要获得一个连续的子串，应使用切片运算符 s[i:j]，它返回字符串 s 中从索引 i(包括 i)到 j(不包括 j)之间的子串，若 i 被省略，Python 就认为 i=0，若 j 被省略，Python 就认为 j=len(s)-1，其中 len(s)表示字符串的长度(稍后介绍)；③要每间隔若干字符返回一个字符，应使用切片运算符 s[i:j:k]，k 为间隔字符的个数，k 为正数表示从左向右截取，k 为负数表示从右向左截取，k 不能为 0。不难想象，s[::-1]获得的将是逆序的整个字符串。

3. Python 字符串的更新

不能企图通过切片运算更新已存在的字符串，如：

```
>>> str='0123456789'
>>> str[0]='a'
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    str[0]='a'
TypeError: 'str' object does not support item assignment
>>> |
```

必须使用 Python 的内建函数 replace()实现字符串的更新，例如：

```
>>> str.replace('0','a')
'a123456789'
>>> |
```

4. Python 字符串中的转义字符

当字符串中需要使用特殊字符时，可以像 C 语言那样，在特殊字符前冠以反斜杠(\)，形成转义字符。Python 的转义字符如表 2-3 所示。

表 2-3　Python 的转义字符

转义字符	描　述
\(在行尾时)	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\0yy	八进制数 yy 代表的字符，例如：\012 代表换行
\xyy	十六进制数 yy 代表的字符，例如：\x0a 代表换行
\other	其他的字符以普通格式输出

5. Python 的字符串运算符

假如 a 变量的值为字符串“Hello”，而 b 变量的值为字符串“Python”，那么，各种运算结果如表 2-4 所示。

表 2-4　Python 的字符串运算符

操作符	描　述	实　例
+	字符串连接	a+b 输出结果：HelloPython
*	重复输出字符串	a*2 输出结果：HelloHello
[]	通过索引获取字符串中的字符	a[1] 输出结果：e
[:]	截取字符串中的一部分(切片)	a[1:4] 输出结果：ell
in	成员运算符——如果字符串中包含给定的字符，则返回 True	‘H’ in a 输出结果：1
not in	成员运算符——如果字符串中不包含给定的字符，则返回 True	‘M’ not in a 输出结果：1
r/R	原始字符串——所有的字符串都是直接按照字面的意思来使用，没有转义为特殊的或不能打印的字符。原始字符串除在字符串的第一个引号前加上字母 r(可以大小写)以外，与普通字符串有着几乎完全相同的语法	print (r‘\n’) 输出结果：\n print (R‘\n’)输出结果：\n
%	格式化字符串	见后文叙述

下面将上述字符串运算放入一个程序中：

```
#!/usr/bin/python3
a = "Hello"
b = "Python"
print("a + b 输出结果：", a + b)
print("a * 2 输出结果：", a * 2)
print("a[1] 输出结果：", a[1])
print("a[1:4] 输出结果：", a[1:4])
if("H" in a) :
   print("H 在变量 a 中")
else :
   print("H 不在变量 a 中")
if("M" not in a) :
   print("M 不在变量 a 中")
else :
   print("M 在变量 a 中")
print(r'\n')
print(R'\n')
```

以上程序的输出结果如下：

```
>>>
a + b 输出结果： HelloPython
a * 2 输出结果： HelloHello
a[1] 输出结果： e
a[1:4] 输出结果： ell
H 在变量 a 中
M 不在变量 a 中
\n
\n
>>>
```

6. Python 字符串的格式化

Python 支持字符串的格式化输出。尽管这样做可能会使表达式非常复杂，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

在 Python 中，字符串格式化的语法与 C 中 printf 函数相似。我们看下面的例子：

```
>>> print ("我叫 %s,今年 %d 岁!" % ('张三', 20))
我叫 张三,今年 20 岁!
>>>
```

Python 字符串格式化用到的符号如表 2-5 所示。

表 2-5 Python 字符串格式化用到的符号

符 号	描 述
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整型数
%u	格式化无符号整型数

续表

符　号	描　述
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数(大写)
%f	格式化浮点数，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	按%f 和%e 的较短者输出
%G	按%f 和%E 的较短者输出
%p	用十六进制数格式化变量的地址

格式化操作符前面可以加入辅助操作符，这一点也与 C 语言极为相似。Python 的格式化辅助操作符如表 2-6 所示。

表 2-6　Python 格式化辅助操作符

符　号	功　能
*	定义宽度或者精度
-	输出左对齐
+	在正数前面显示加号(+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零(‘0’)，在十六进制数前面显示‘0x’或者‘0X’(取决于用的是‘x’还是‘X’)
0	显示的数字前面填充‘0’而不是默认的空格
%	‘%%’输出一个单一的‘%’
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度，n 是小数点后的位数(如果可用的话)

尽管使用格式化操作符输出字符串很方便，而且与 C 语言兼容，但 Python 并不推荐使用这种方法，具体原因参见本书 4.1.2 小节。

7. Python 的三引号

Python 的三引号(三个单引号或三个双引号均可)用于字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。举例如下：

```
>>> para_str = """这是Python多行字符串的实例
多行字符串可以使用制表符
TAB ( \t )。
也可以使用换行符 [ \n ]。
"""

>>> print (para_str)
这是Python多行字符串的实例
多行字符串可以使用制表符
TAB (     )。
也可以使用换行符 [
 ]。

>>> |
```

三引号把程序员从引号和特殊字符串的泥潭里解脱出来，自始至终保持一小块字符串的格式，符合所见即所得(What You See Is What You Get，WYSIWYG)的特征。

一个典型的应用是，当需要一段 HTML 或者 SQL 时，如果用字符串组合转义字符，将会非常繁琐，而使用三引号不失为一种绝佳的选择，例如：

```
errHTML = '''
<HTML><HEAD><TITLE>
Friends CGI Demo</TITLE></HEAD>
<BODY><H3>ERROR</H3>
<B>%s</B><P>
<FORM><INPUT TYPE=button VALUE=Back
ONCLICK="window.history.back()"></FORM>
</BODY></HTML>
'''
cursor.execute('''
CREATE TABLE users (
login VARCHAR(8),
uid INTEGER,
prid INTEGER)
''')
```

三引号中可以使用成对出现的字符串定界符，例如：

```
str5='''I want a book,and you want a book,\n too.'''    #三单引号
str6='''"I want a book,and you want a book,\n too."'''
  #三单引号中间使用双引号

str7='''I want a book,
and you want a book,\n too.'''   #三单引号中有换行符
str8="""I want a book,
and you want a book,\n too."""   #三双引号中有换行符
print(str5); print(str6); print(str7); print(str8)
```

上面程序段的运行结果如下：

```
I want a book,and you want a book,
 too.
"I want a book,and you want a book,
 too."
I want a book,
and you want a book,
 too.
I want a book,
and you want a book,
 too.
```

注意： Python 字符串的几种定界符均已介绍完毕，现对其归纳如下。

① 单引号中可以使用双引号，中间的会当作字符串输出。

② 双引号中可以使用单引号，中间的会当作字符串输出。

③ 三单引号和三双引号中间的字符串在输出时保持原来的格式。

④ 单引号和双引号不能搭配使用，必须成对使用。

8. Unicode 字符串

在 Python 2 中，普通字符串是以 8 位 ASCII 码形式进行存储的，而 Unicode 字符串则存储为 16 位 Unicode 字符串。使用 Unicode 字符串，旨在表示更多的字符，其语法非常简单，只要在字符串前面加上前缀 u 即可。

Python 3 中默认所有的字符串都为 Unicode 的，因而不必在字符串的前面加前缀 u。

9. 原始字符串

前面讲解字符串运算符时已提及原始字符串并举例(表 2-4)，此处再强调一下。在字符串前面加字母 r 或 R，就是告诉 Python 解释器，其后的字符串是原始字符串(Raw String)。原始字符串指的是，字符串内的所有字符均保持其原有的含义，不做转义处理。换言之，“\n”在原始串中是两个字符，即“\”和“n”，而不会被转义为换行符。由于正则表达式(本章最后介绍)经常与“\”冲突，所以，当一个字符串中使用了正则表达式之后，往往在前面加字母 r，具体例子参见 2.8.2 节。此外，在描述文件路径时，往往使用“\”，为避免歧义，也往往在其前面冠以字母 r，具体例子见 5.1 节。

10. Python 的字符串内建函数

Python 的字符串有很多内建函数，常用的如表 2-7 所示。

表 2-7　常用的 Python 字符串内建函数

序　号	方法及描述
1	capitalize() 将字符串的第一个字符转换为大写
2	center(width, fillchar) 返回一个指定的宽度 width 居中的字符串，fillchar 为填充的字符，默认为空格
3	count(str, beg=0, end=len(string)) 返回 str 在 string 里面出现的次数，如果指定 beg 或 end，则返回指定范围内 str 出现的次数
4	bytes.decode(encoding=“utf-8”, errors=“strict”) Python 3 中没有 decode 方法，但我们可以使用 bytes 对象的 decode()方法来解码给定的 bytes 对象，这个 bytes 对象可以由 str.encode()来编码返回
5	encode(encoding=‘utf-8’, errors=‘strict’) 以 encoding 指定的编码格式编码字符串，如果出错，则默认报一个 ValueError 的异常，除非 errors 指定的是‘ignore’或者‘replace’
6	endswith(suffix, beg=0, end=len(string)) 检查字符串是否以 suffix 结束，如果指定 beg 或者 end，则检查指定的范围内是否以 suffix 结束，如果是，则返回 True，否则返回 False
7	expandtabs(tabsize=8) 把字符串中的 Tab 符号转为空格，Tab 符号默认的空格数是 8
8	find(str, beg=0, end=len(string)) 检测 str 是否包含在字符串中，如果用 beg 和 end 指定范围，则检查是否包含在指定的范围内，如果是，则返回开始的索引值，否则返回-1

续表

序　号	方法及描述
9	index(str, beg=0, end=len(string)) 跟 find()方法一样，只不过如果 str 不在字符串中，则会报一个异常
10	isalnum() 如果字符串至少有一个字符，并且所有字符都是字母或数字，则返回 True，否则返回 False
11	isalpha() 如果字符串至少有一个字符，并且所有字符都是字母，则返回 True，否则返回 False
12	isdigit() 如果字符串只包含数字，则返回 True，否则返回 False
13	islower() 如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
14	isnumeric() 如果字符串中只包含数字字符，则返回 True，否则返回 False
15	isspace() 如果字符串中只包含空格，则返回 True，否则返回 False
16	istitle() 如果字符串是标题化的(见 title())，则返回 True，否则返回 False
17	isupper() 如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
18	join(seq) 以指定字符串作为分隔符，将 seq 中所有的元素(字符串表示)合并为一个新的字符串
19	len(string) 返回字符串长度，即字符串中字符的个数
20	ljust(width[, fillchar]) 返回一个原字符串左对齐，并使用 fillchar 填充至长度 width 的新字符串，fillchar 默认为空格
21	lower() 转换字符串中所有大写字母为小写
22	lstrip() 截掉字符串左边的空格
23	maketrans(intab, outtab) 创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串，表示转换的目标
24	max(str) 返回字符串 str 中最大的字母
25	min(str) 返回字符串 str 中最小的字母

续表

序　号	方法及描述
26	replace(old, new [, max]) 将字符串中的 old 替换成 new。如果 max 指定，则替换不超过 max 次
27	rfind(str, beg=0, end=len(string)) 类似于 find()函数，不过是从右边开始查找
28	rindex(str, beg=0, end=len(string)) 类似于 index()，不过是从右边开始
29	rjust(width [, fillchar]) 返回一个原字符串右对齐，并使用 fillchar(默认空格)填充至长度 width 的新字符串
30	rstrip() 删除字符串末尾的空格
31	split(str= “”, num=string.count(str)) num=string.count(str))以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num 个子字符串
32	splitlines([keepends]) 按照行(‘\r’, ‘\r\n’, ‘\n’)分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符
33	startswith(str, beg=0, end=len(string)) 检查字符串是否是以 str 开头，是则返回 True，否则返回 False。如果 beg 和 end 指定值，则在指定范围内检查
34	strip([chars]) 在字符串上执行 lstrip()和 rstrip()
35	swapcase() 将字符串中大写转换为小写，小写转换为大写
36	title() 返回“标题化”的字符串，就是说，所有单词都是以大写开始，其余字母均为小写(见 istitle())
37	translate(table, deletechars=“”) 根据 table 给出的表(包含 256 个字符)转换 string 的字符, 需要过滤掉的字符放到 deletechars 参数中
38	upper() 转换字符串中的小写字母为大写
39	zfill(width) 返回长度为 width 的字符串，原字符串右对齐，前面填充 0
40	isdecimal() 检查字符串是否只包含十进制字符，如果是则返回 True，否则返回 False

这些内建函数为我们处理字符串带来了极大的方便。读者学习 Python 语言时，不要试图自己编写程序来处理字符串，应当充分利用这些内建函数完成相应的功能。

细心的读者可能已经注意到，表 2-7 的标题上写的是“函数”，而表格栏目中谓之“方法”。其实，函数与方法并无严格的区别。在面向过程的语言中，一个模块主要强调的是数据处理，就像数学上的函数一样，故称为函数；在面向对象的语言中，一般把类中

定义的函数称作方法、服务或操作，因为它主要强调这个类的对象封装了一些属性和方法(变量和函数)并向外提供服务。表 2-7 中所列函数均是定义在某个类里面的，故称为方法。说到底，方法就是类函数。

有鉴于此，在不产生混淆的前提下，本书的后续章节拟不严格区分函数与方法。

有关类与面向对象的概念，读者可参阅第 6 章中的相关内容。

下面看几个关于字符串操作的例子(其中使用了下一节将要介绍的列表)：

```
>>> myString = "+"
>>> seq = ['1', '2','3']
>>> myString.join(seq)
'1+2+3'
>>> myString = "This is an apple and that is an apple too."
>>> myString.replace('apple', 'orange', myString.count('apple'))
'This is an orange and that is an orange too.'
>>> myString.replace('apple', 'orange', 1)
'This is an orange and that is an apple too.'
>>> myString.split(' ', 3)
['This', 'is', 'an', 'apple and that is an apple too.']
>>> |
```

2.4　列表与序列

在介绍列表之前，我们先了解一下 Python 中“序列”的概念。在 Python 中，序列是最基本的数据结构。序列中的每个元素都被分配一个数字，以表明它的位置，并称为索引。其中，第一个索引值为 0，第二个索引值为 1，依此类推。

Python 中的序列都可以进行索引、切片、加、乘、检查成员等操作。为了使用方便，Python 还内建了确定序列长度以及确定最大和最小元素的方法。

Python 中最常用的序列是列表和元组，本节介绍列表，元组放在下一节介绍。

1. 列表的概念

列表(list)是 Python 中使用最频繁的数据类型，它是放在方括号[]内、用逗号分隔的一系列元素。

列表中，元素的类型可以不同，支持数字、字符串，甚至可以包含列表。换言之，列表允许嵌套。

2. 列表的创建

创建一个列表时，只要把逗号分隔的不同的数据项用方括号括起来即可。例如：

```
list1 = ['Google', 'Runoob', 1997, 2017]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

与字符串一样，列表同样可以被索引、截取和组合。列表被截取后，返回一个包含所需元素的新列表。与字符串的索引一样，列表索引也是从 0 开始的。

3. 列表的访问与截取

可以使用下标索引来访问列表中的元素，同样，也可以使用类似于字符串切片运算的

形式截取列表中的元素，例如：

```
>>> list1 = ['Google', 'Runoob', 1997, 2017]
>>> list2 = [1, 2, 3, 4, 5, 6, 7 ]
>>> print ("list1[0]: ", list1[0])
list1[0]:  Google
>>> print ("list2[1:5]: ", list2[1:5])
list2[1:5]:  [2, 3, 4, 5]
>>> |
```

令：

```
L=['Google', 'Runoob', 'Taobao']
```

则截取操作如表 2-8 所示。

表 2-8　Python 列表的截取操作

Python 表达式	结　果	描　述
L[2]	'Taobao'	读取第三个元素
L[-2]	'Runoob'	从右侧开始读取倒数第二个元素
L[1:]	['Runoob', 'Taobao']	输出从第二个元素开始的所有元素

下面将上述列表截取操作放入一个程序中：

```
#!/usr/bin/python3
L=['Google', 'Runoob', 'Taobao']
print(L[2])
print(L[-2])
print(L[1:])
```

以上程序的输出结果如下：

```
Taobao
Runoob
['Runoob', 'Taobao']
>>>
```

注意： 字符串、列表和元组(下节介绍)三者都是序列，都支持切片运算，操作方法也极为相似，只是操作结果的类型有所不同。此外，不能通过切片运算对字符串和元组进行更新，列表则可以。

4. 列表的更新

可以对列表的数据项进行修改或更新，也可以使用 append()方法(稍后介绍)添加一些列表项。例如：

```
>>> list = ['Google', 'Runoob', 1997, 2017]
>>> print ("第三个元素为 : ", list[2])
第三个元素为 :  1997
>>> list[2] = 2001
>>> print ("更新后的第三个元素为 : ", list[2])
更新后的第三个元素为 :  2001
>>> |
```

5. 列表元素的删除

可以使用 del 语句来删除列表中的元素，例如：

```
>>> list = ['Google', 'Runoob', 1997, 2017]
>>> print (list)
['Google', 'Runoob', 1997, 2017]
>>> del list[2]
>>> print ("删除第三个元素后 : ", list)
删除第三个元素后 :  ['Google', 'Runoob', 2017]
>>> |
```

使用 remove()方法也可以删除列表的元素，我们稍后再讨论。

6. 列表操作符

列表对 + 和 * 的操作符与字符串相似。+ 号用于组合列表，* 号用于重复列表。+ 和 * 的用法如表 2-9 所示。

表 2-9　列表中+和*的用法

Python 表达式	结　果	描　述
len([1, 2, 3])	3	长度(即列表中元素的个数)
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合(即拼接)
[‘Hi!’] * 4	[‘Hi!’, ‘Hi!’, ‘Hi!’, ‘Hi!’]	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print (x,end=‘ ’)	1 2 3	遍历并输出列表的各个元素

注意： 为了让列表的各个元素打印在一行上，而且相邻两个元素之间隔一个空格，表 2-9 的最后一行使用了 print (x,end=‘ ’)这种打印格式，后面遍历元组和集合时采用了同样的方法，3.4.6 节对此将做详解。

7. 列表嵌套

列表嵌套指的是在列表里创建其他列表，例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
>>> |
```

8. Python 列表中的内建函数与方法

Python 列表中的内建函数如表 2-10 所示。

表 2-10　Python 列表中的内建函数

序　号	函　数
1	len(list)　返回列表元素的个数
2	max(list)　返回列表元素的最大值
3	min(list)　返回列表元素的最小值
4	list(seq)　将元组、字典、集合、字符串等转换为列表

Python 列表中的方法如表 2-11 所示。

表 2-11　Python 列表中的方法

序　号	方　法
1	list.append(obj)　在列表末尾添加新的对象
2	list.count(obj)　统计某个元素在列表中出现的次数
3	list.extend(seq)　在列表末尾一次性追加另一个序列中的多个值(用新列表扩展原来的列表)
4	list.index(obj)　从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj)　将对象插入列表
6	list.pop(obj=list[-1])　移除列表中的一个元素(默认最后一个元素)，并且返回该元素的值
7	list.remove(obj)　移除列表中某个值的第一个匹配项
8	list.reverse()　将列表中的元素反向
9	list.sort([func])　对原列表进行排序
10	list.clear()　清空列表
11	list.copy()　复制列表

下面看几个列表操作的例子：

```
>>> #向列表尾部添加元素
>>> a=[1,2,3,4]
>>> a.append(5)
>>> print(a)
[1, 2, 3, 4, 5]
>>> 
```

```
>>> #插入一个元素
>>> a=[1,2,4]
>>> a.insert(2,3)
>>> print(a)
[1, 2, 3, 4]
>>> 
```

```
>>> #扩展列表
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4, 5, 6]
>>> 
```

```
>>> #删除元素（如有重复元素，只会删除最靠前的）
>>> a=[1,2,3,2]
>>> a.remove(2)
>>> print(a)
[1, 3, 2]
>>> 
```

```
>>> #删除指定位置的元素，默认为最后一个元素
>>> a=[1, 2, 3, 4, 5, 6]
>>> a.pop()
6
>>> print(a)
[1, 2, 3, 4, 5]
>>> a.pop(3)
4
>>> print(a)
[1, 2, 3, 5]
>>> |
>>> #逆序
>>> a=[1, 2, 3, 4, 5, 6]
>>> a.reverse()
>>> print(a)
[6, 5, 4, 3, 2, 1]
>>> |
>>> #排序
>>> a=[2,4,7,6,3,1,5]
>>> a.sort()
>>> print(a)
[1, 2, 3, 4, 5, 6, 7]
>>> |
```

2.5 元　组

Python 的元组(tuple)与列表相似，不同之处在于元组的元素是不能修改的。另外，列表使用方括号[]，而元组使用圆括号()。

1. 元组的创建

元组的创建很简单，只需要在括号中添加元素，并使用逗号分隔各元素即可。例如：

```
tup1 = ('Google', 'Runoob', 1997, 2017)
tup2 = (1, 2, 3, 4, 5 )
tup3 = ("a", "b", "c", "d")
```

创建空元组时，使用如下语句：

```
tup1 = ()
```

注意：

- 当元组中只包含一个元素时，需要在元素后面添加逗号，例如 tup1 = (50,)。
- 与字符串类似，元组的下标索引也从 0 开始，而且也可以进行截取、组合等操作。

2. 元组的访问

可以使用下标索引来访问元组中的元素，例如：

```
>>> tup1 = ('Google', 'Runoob', 1997, 2017)
>>> tup2 = (1, 2, 3, 4, 5, 6, 7 )
>>> print ("tup1[0]: ", tup1[0])
tup1[0]:  Google
>>> print ("tup2[1:5]: ", tup2[1:5])
tup2[1:5]:  (2, 3, 4, 5)
>>>
```

3. 元组的修改

如前所述，元组中的元素值是不允许修改的，但可以对元组进行连接组合。假如：

```
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
```

以下修改元组元素的操作是非法的：

```
>>> tup1[0] = 100
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    tup1[0] = 100
TypeError: 'tuple' object does not support item assignment
>>>
```

但可以通过连接运算符“+”创建一个新的元组：

```
>>> tup3 = tup1 + tup2
>>> print (tup3)
(12, 34.56, 'abc', 'xyz')
>>>
```

4. 元组的删除

既然元组不能修改，那么其中的元素值也就不允许删除，但我们可以使用 del 语句删除整个元组，例如：

```
>>> tup = ('Google', 'Runoob', 1997, 2017)
>>> print (tup)
('Google', 'Runoob', 1997, 2017)
>>> del tup
>>> print (tup)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print (tup)
NameError: name 'tup' is not defined
>>>
```

可见，元组被删除后，再输出该元组时会显示异常信息。

5. 元组运算符

与字符串类似，元组之间可以使用 + 号和 * 号进行运算。这就意味着可以将它们组合和复制，运算后将生成一个新的元组。

表 2-12 给出了几个例子。

表 2-12　Python 元组的运算符

Python 表达式	结　果	描　述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	重复
3 in (1, 2, 3)	True	元素是否存在于元组中
for x in (1, 2, 3): print (x,end=' ')	1 2 3	遍历并输出元组的各个元素

6. 元组的索引、截取

前已提及，元组是一个序列，所以可以访问元组中指定位置的元素，也可以通过索引截取元组中的一段元素。假设元组为 L = ('Google', 'Taobao', 'Runoob')，表 2-13 说明了元组的索引和截取。

表 2-13　Python 元组的索引和截取

Python 表达式	结　果	描　述
L[2]	'Runoob!'	读取第三个元素
L[-2]	'Taobao'	反向读取；读取倒数第二个元素
L[1:]	('Taobao', 'Runoob!')	截取元素，从第二个开始后的所有元素

运行结果如下：

```
>>> L = ('Google', 'Taobao', 'Runoob')
>>> L[2]
'Runoob'
>>> L[-2]
'Taobao'
>>> L[1:]
('Taobao', 'Runoob')
>>> |
```

7. 元组的内建函数

Python 元组包含了一些内建函数，如表 2-14 所示。

表 2-14　Python 元组的内建函数

序号	方法及描述	举　例

1	len(tuple)
	计算元组中元素的个数
		`>>> tuple1 = ('Google', 'Runoob', 'Taobao')`
		`>>> len(tuple1)`
		`3`
		`>>> `

2	max(tuple)
	返回元组中元素的最大值
		`>>> tuple2 = ('5', '4', '8')`
		`>>> max(tuple2)`
		`'8'`
		`>>> `

续表

序号	方法及描述	举　例

3	min(tuple)
	返回元组中元素的最小值
		`>>> tuple2 = ('5', '4', '8')`
		`>>> min(tuple2)`
		`'4'`
		`>>>`

4	tuple(seq)
	将列表、字符串、字典、集合等转换为元组
		`>>> list1= ['Google', 'Taobao', 'Runoob', 'Baidu']`
		`>>> tuple1=tuple(list1)`
		`>>> tuple1`
		`('Google', 'Taobao', 'Runoob', 'Baidu')`
		`>>>`

2.6　字　　典

1. 字典的概念

Python 中的字典(dictionary)是一种可变容器模型，且可以存储任意类型的对象。

字典中的每个项都是“键/值对”(有时写作“键-值对”或“键值对”)，“键”与“值”之间用冒号(:)分隔，而每个“对”之间用逗号(,)分隔，整个字典放在花括号{}中，格式如下：

```
d = {key1 : value1, key2 : value2 }
```

对每个键/值对而言，键必须是唯一的，但值可以改变。值可以取任何数据类型，但键必须是不可变的。例如，字符串、数字或元组均可作为键，但列表不可以。

2. 字典的创建

Python 中创建字典的方法很简单，只要将键值对放入花括号内，并用逗号隔开即可。一个简单的字典例子如下：

```
dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

也可这样创建字典：

```
dict1 = { 'abc': 456 }
dict2 = { 'abc': 123, 98.6: 37 }
```

3. 字典的访问

把相应的键放入方括号内，即可得到相应的值。例如：

```
>>> dict1 = {'Name':'Runoob','Age':7,'Class':'First'}
>>> print("dict1['Name']",dict1['Name'])
dict1['Name'] Runoob
>>> print("dict1['Age']",dict1['Age'])
dict1['Age'] 7
>>>
```

如果所用的键在字典中不存在，则数据无法访问，会输出错误信息，例如：

```
>>> dict1 = {'Name':'Runoob','Age':7,'Class':'First'}
>>> print("dict1['Alice']",dict1['Alice'])

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print("dict1['Alice']",dict1['Alice'])
KeyError: 'Alice'
>>> |
```

4. 字典的添加与修改

向字典添加新的键/值对，便向字典中添加了新的内容。修改或添加已有键/值对的例子如下：

```
>>> dict1 = {'Name':'Runoob','Age':7,'Class':'First'}
>>> dict1['Age'] = 8                       #更新age
>>> dict1['School'] = "TJPU"               #添加信息
>>> print("dict1['Age']:",dict1['Age'])
dict1['Age']: 8
>>> print("dict1['School']:",dict1['School'])
dict1['School']: TJPU
>>> |
```

5. 字典元素的删除

既能删除字典中的单一元素，也能将整个字典清空，而且清空只需一项操作。

删除一个字典用 del 命令，例如：

```
>>> dict1 = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}
>>> del dict1['Name'] # 删除键 'Name'
>>> dict1
{'Class': 'First', 'Age': 7}
>>> dict1.clear()      # 删除字典元素
>>> dict1
{}
>>> print ("dict1['Age']: ", dict1['Age'])
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    print ("dict1['Age']: ", dict1['Age'])
KeyError: 'Age'
>>> del dict1          # 删除字典
>>> print ("dict1['Age']: ", dict1['Age'])
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print ("dict1['Age']: ", dict1['Age'])
NameError: name 'dict1' is not defined
>>> dict1
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    dict1
NameError: name 'dict1' is not defined
>>> |
```

第一次引发异常是因为字典中已无元素(空字典)，试图访问键‘age’会出错；第二次引发异常是因为前面已执行了 del 操作，字典已不复存在，当然不能访问键‘age’；第三次引发异常也是因为字典不存在，所以两次引发异常的名称均为 NameError。

6. 字典键的特性

字典的值可以取任何 Python 对象，没有任何限制，它既可以是标准对象，也可以是用户自己定义的对象，但键不行。

关于字典的键，有两点必须牢记。

(1) 同一个键不得出现两次。创建字典时，如果同一个键被赋值两次，则后一个值被记住，例如：

```
>>> dict1 = {'Name':'Runoob','Age':7,'Name':'小菜鸟'}
>>> print("dict1['Name']",dict1['Name'])
dict1['Name'] 小菜鸟
>>> |
```

(2) 键必须不可变。可以用数字、字符串或元组做键，用列表则不行，例如：

```
>>> dict1 = {['Name']:'Runoob','Age':7}
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    dict1 = {['Name']:'Runoob','Age':7}
TypeError: unhashable type: 'list'
>>> |
```

7. 字典的内建函数与方法

Python 字典包含的内建函数如表 2-15 所示。

表 2-15　Python 字典的内建函数

序号	函数及描述	举　例

1	len(dict)
	计算字典元素个数，即键的总数
		`>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}`
		`>>> len(dict)`
		`3`

2	str(dict)
	输出字典，以可打印的字符串表示
		`>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}`
		`>>> str(dict)`
		`"{'Name': 'Runoob', 'Class': 'First', 'Age': 7}"`

3	type(variable)
	返回输入的变量类型，如果变量是字典就返回字典类型
		`>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}`
		`>>> type(dict)`
		`<class 'dict'>`

Python 字典包含的内建方法如表 2-16 所示。

表 2-16　Python 字典的内建方法

序号	函数及描述
1	radiansdict.clear() 删除字典内所有元素
2	radiansdict.copy() 返回一个字典的浅拷贝
3	radiansdict.fromkeys(seq[, val]) 创建一个新字典，以序列 seq 中的元素做字典的键，val 为字典所有键对应的初始值(缺省时为 None)

续表

序号	函数及描述
4	radiansdict.get(key, default=None) 返回指定键的值，如果值不在字典中，则返回 default 值
5	key in dict 如果键在字典 dict 里，则返回 True，否则返回 False
6	radiansdict.items() 以列表返回可遍历的(键, 值)元组数组
7	radiansdict.keys() 以列表返回一个字典所有的键
8	radiansdict.setdefault(key, default=None) 与 get()类似，但如果键不存在于字典中，将会添加键并将值设为 default
9	radiansdict.update(dict2) 把字典 dict2 的键/值对更新到字典里
10	radiansdict.values() 以列表返回字典中的所有值

下面看几个典型的例子：

```
>>> #返回一个具有相同键-值对的新字典（浅复制）
>>> x={'name':'admin','machines':['foo','bar','bax']}
>>> y=x.copy()
>>> y['name']='ylh' #替换值，原字典不受影响
>>> y['machines'].remove('bar') #修改了某个值(原地修改不是替换)，原字典会改变
>>> y
{'name': 'ylh', 'machines': ['foo', 'bax']}
>>> x
{'name': 'admin', 'machines': ['foo', 'bax']}
>>> |
>>> #使用给定的键建立新的字典，每个键默认的对应的值为none
>>> {}.fromkeys(['name','sex','age'])
{'name': None, 'age': None, 'sex': None}
>>> dict.fromkeys(['name','sex','age'])
{'name': None, 'age': None, 'sex': None}
>>> dict.fromkeys(['name','sex','age'],'(unknown)')
{'name': '(unknown)', 'age': '(unknown)', 'sex': '(unknown)'}
>>> |
>>> #访问指定键的值
>>> d={}
>>> print (d['name'])  #键不存在，出错
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    print (d['name'])  #键不存在，出错
KeyError: 'name'
>>> print (d.get('name'))
None
>>> d.get('name','N/A')
'N/A'
>>> d['name']='Eric'
>>> d.get('name')
'Eric'
>>> |
```

```
>>> #利用一个字典更新另外一个字典
>>> d={'x':1,'y':2,'z':3}
>>> f={'y':5}
>>> d.update(f)
>>> d
{'x': 1, 'z': 3, 'y': 5}
>>> 
```

2.7 集　　合

1. 集合概述

集合(set)是 Python 的基本数据类型，把不同的元素组合在一起便形成了集合。组成一个集合的成员称作该集合的元素(element)。在一个集合中不能有相同的元素。下面是集合的例子：

```
>>> li=['a','b','c','a']
>>> se =set(li)
>>> se
{'b', 'c', 'a'}
>>> 
```

可以看到，相同的元素被自动删除了。

集合对象是一组无序排列的可哈希的值，集合成员可以作为字典的键。相比之下，列表对象是不可哈希的，所以下面的程序会出错：

```
>>> li=[['a','b','c'],['a','c']]
>>> se = set(li)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    se = set(li)
TypeError: unhashable type: 'list'
>>>
```

集合可以分为两类：可变集合(set)与不可变集合(frozenset)。

可变集合可添加和删除元素，是非可哈希的，不能用作字典的键，也不能做其他集合的元素。而不可变集合与之相反。

2. 集合操作符和关系符号

集合有各种操作，各种操作符和关系符号如表 2-17 所示。

表 2-17　集合操作符和关系符号

数学符号	Python 符号	说　明
$\in$	in	是……的成员
$\notin$	not in	不是……的成员
=	==	等于
$\neq$	!=	不等于
$\subset$	<	是……的真子集
$\subseteq$	<=	是……的子集

续表

数学符号	Python 符号	说　明
⊃	>	是……的真超集
⊇	>=	是……的超集
∩	&	交集
∪	|	并集
-或\	-	差集或相对补集
Δ	^	对称差分

3. 集合的相关操作

(1) 集合的创建。

由于集合没有自己的语法格式，只能通过集合的工厂方法 set()和 frozenset()来创建：

```
>>> s = set('beginman')
>>> s
{'g', 'b', 'a', 'i', 'n', 'm', 'e'}
>>> t = frozenset('pythonman')
>>> t
frozenset({'y', 'h', 'a', 'p', 't', 'n', 'o', 'm'})
>>> type(s),type(t)
(<class 'set'>, <class 'frozenset'>)
>>> len(s),len(t)
(7, 8)
>>> s==t
False
>>> s=t
>>> s==t
True
>>>
```

(2) 集合的访问。

由于集合本身是无序的，所以不能像列表和元组那样，为集合创建索引或进行切片操作，只能循环遍历或使用 in、not in 来访问或判断集合元素(有关循环的内容将在下一章中做具体介绍)：

```
>>> 'a' in s
True
>>> 'z' in s
False
>>> for i in s:
        print (i,' ',end='')

y  h  a  p  t  n  o  m
>>>
```

(3) 集合的更新。

Python 内建了以下方法，可以实现集合的更新：

```
s.add()
s.update()
s.remove()
```

当然，只有可变集合才能更新，试图更新不可变集合将会出错。例如：

```
>>> s.add(0)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    s.add(0)
AttributeError: 'frozenset' object has no attribute 'add'
>>> type(s)
<class 'frozenset'>
>>> se = set(s)
>>> se
{'y', 'h', 'a', 'p', 't', 'n', 'o', 'm'}
>>> type(se)
<class 'set'>
>>> se.add(0)
>>> se
{0, 'y', 'h', 'a', 'p', 't', 'n', 'o', 'm'}
>>> se.update('MM')
>>> se
{0, 'y', 'h', 'a', 'p', 't', 'n', 'o', 'm', 'M'}
>>> se.update('Django')
>>> se
{0, 'g', 'j', 'y', 'h', 'a', 'p', 't', 'n', 'o', 'm', 'M', 'D'
>>> se.remove('D')
>>> se
{0, 'g', 'j', 'y', 'h', 'a', 'p', 't', 'n', 'o', 'm', 'M'}
>>>
```

内建的 del 命令可以删除集合本身。

4. 集合类型操作符

集合类型操作符有 7 类。

(1) in、not in(是否是集合的元素)。

(2) ==、!=(集合等价与不等价)。

(3) 子集、超集(见表 2-17)。例如：

```
>>> set('shop')<set('cheeshop')
True
>>> set('bookshop')>=set('shop')
True
>>>
```

(4) 联合(|)。

联合(union)操作与集合的 or 操作其实是等价的，联合操作符还有一个与之等价的方法 union()。例如：

```
>>> s1=set('begin')
>>> s2=set('man')
>>> s3=s1|s2
>>> s3
{'a', 'i', 'e', 'n', 'm', 'b', 'g'}
>>> s1.union(s2)
{'a', 'i', 'e', 'n', 'm', 'b', 'g'}
```

但+运算则不可用于集合：

```
>>> s3New = s1+s2
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    s3New = s1+s2
TypeError: unsupported operand type(s) for +: 'set' and 'set'
>>>
```

(5) 交集(&)。

与集合的 and 操作等价，交集符号&的等价方法是 intersection()。例如：

```
>>> s1&s2
{'n'}
>>> s1.intersection(s2)
{'n'}
>>>
```

(6) 差补(-)。

与之等价的方法是 difference()。

```
>>> s1-s2
{'b', 'g', 'i', 'e'}
>>> s1.difference(s2)
{'b', 'g', 'i', 'e'}
>>>
```

(7) 对称差分(^)。

对称差分是集合的 xor(异或)，取得的元素属于 s1 和 s2，但不同时属于 s1 和 s2，其等价的方法是 symmetric_difference()。例如：

```
>>> s1=set('begin')
>>> s2=set('man')
>>> s1^s2
{'i', 'g', 'a', 'e', 'b', 'm'}
>>> s1.symmetric_difference(s2)
{'i', 'g', 'a', 'e', 'b', 'm'}
>>>
```

注意集合之间的 and、or 运算：

```
>>> s1 and s2
{'n', 'a', 'm'}
>>> s1 or s2
{'e', 'i', 'b', 'n', 'g'}
>>>
```

可见，s1 and s2 运算取的是 s2，而 s1 or s2 运算取的是 s1。

5. 集合转换为字符串、元组

举例如下：

```
>>> str(s1)
"{'e', 'i', 'b', 'n', 'g'}"
>>> tuple(s1)
('e', 'i', 'b', 'n', 'g')
>>>
```

6. 关于集合的内建函数、内建方法

(1) len()：返回集合元素的个数。

(2) set()、frozenset()：创建集合(属工厂函数)。

(3) 适合所有集合的方法(见表 2-18)。

表 2-18　适合所有集合的方法

方法名称	操　作
s.issubset(t)	如果 s 是 t 的子集，则返回 True，否则返回 False
s.issuperset(t)	如果 s 是 t 的超集，则返回 True，否则返回 False
s.union(t)	返回一个新集合，该集合是 s 和 t 的并集
s.intersection(t)	返回一个新集合，该集合是 s 和 t 的交集
s.difference(t)	返回一个新集合，该集合是 s 的成员，但不是 t 的成员
s.symmetric_difference(t)	返回一个新集合，该集合是 s 或 t 的成员，但不是 s 和 t 共有的成员
s.copy()	返回一个新集合，它是集合 s 的浅拷贝

例如：

```
>>> s=set('cheeseshop')
>>> t=set('bookshop')
>>> s
{'p', 'h', 'e', 's', 'o', 'c'}
>>> t
{'p', 'h', 's', 'o', 'b', 'k'}
>>> s.issubset(t)
False
>>> s.issuperset(t)
False
>>> s.union(t)
{'p', 'h', 'e', 's', 'o', 'b', 'k', 'c'}
>>> s.intersection(t)
{'h', 'p', 's', 'o'}
>>> s.difference(t)
{'e', 'c'}
>>> s.symmetric_difference(t)
{'e', 'b', 'k', 'c'}
>>> s.copy()
{'e', 'p', 's', 'o', 'h', 'c'}
>>>
```

(4)　仅适合可变集合的方法(见表 2-19)。

表 2-19　仅适合可变集合的方法

方法名称	操　作
s.update(t)	用 t 中的元素修改 s，即 s 现在包含 s 或 t 的成员
s.intersection_update(t)	s 中的成员是共同属于 s 和 t 中的元素
s.difference_update(t)	s 中的成员是属于 s 但不包含在 t 中的元素
s.symmetric_difference_update(t)	s 中的成员更新为那些包含在 s 或 t 中，但不是 s 和 t 共有的元素
s.add(obj)	在集合 s 中添加对象 obj
s.remove(obj)	从集合 s 中删除对象 obj，如果 obj 不是集合 s 中的元素(obj not in s)，将引发 KeyError 错误
s.discard(obj)	如果 obj 是集合 s 中的元素，从集合 s 中删除对象 obj
s.pop()	删除集合 s 中的任意一个对象，并返回它
s.clear()	删除集合 s 中的所有元素

例如：

```
>>> s.update(t)
>>> s
{'p', 'h', 'e', 's', 'o', 'b', 'k', 'c'}
>>> s=set('cheeseshop')
>>> t=set('bookshop')
>>> s.intersection_update(t)
>>> s
{'h', 'p', 's', 'o'}
>>> s=set('cheeseshop')
>>> t=set('bookshop')
>>> s.difference_update(t)
>>> s
{'e', 'c'}
>>> s=set('cheeseshop')
>>> t=set('bookshop')
>>> s.symmetric_difference_update(t)
>>> s
{'e', 'b', 'k', 'c'}
>>> s.add('o')
>>> s
{'e', 'o', 'b', 'k', 'c'}
>>> s.remove('b')
>>> s
{'e', 'o', 'k', 'c'}
>>> s.remove('a')
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    s.remove('a')
KeyError: 'a'
```

‘a’元素已不在集合中，试图删除将引发异常。又如：

```
>>> s.discard('a')
>>> s
{'e', 'o', 'k', 'c'}
>>> s.discard('e')
>>> s
{'o', 'k', 'c'}
>>> s.pop()
'o'
>>> s
{'k', 'c'}
>>> s.clear()
>>> s
set()
>>> |
```

2.8　正则表达式

首先须指出，正则表达式并不是 Python 的一部分。正则表达式是用于处理字符串的强大工具，它拥有自己独特的语法以及一个独立的处理引擎，效率上可能不如 str 自带的方法，但功能十分强大。Python 中内建了一个 re 模块以支持正则表达式。

正则表达式有两种基本操作，分别是匹配和替换。

匹配指的是在一个文本字符串中搜索并匹配一个特殊表达式；替换指的是在一个字符

串中查找并替换一个特殊表达式的字符串。

2.8.1　基本元素

正则表达式定义了一系列的特殊字符元素，以执行匹配操作。表 2-20 列示了正则表达式的基本字符。

表 2-20　正则表达式的基本字符

字　符	描　述
text	匹配 text 字符串
.	匹配除换行符之外的任意一个单个字符
^	匹配一个字符串的开头
$	匹配一个字符串的末尾

在正则表达式中，还可用匹配限定符来约束匹配的次数。表 2-21 列示了正则表达式的匹配限定符。

表 2-21　正则表达式的匹配限定符

最大匹配	最小匹配	描　述
*	*?	重复匹配前表达式零次或多次
+	+?	重复匹配前表达式一次或多次
?	??	重复匹配前表达式零次或一次
{m}	{m}?	精确重复匹配前表达式 m 次
{m,}	{m,}?	至少重复匹配前表达式 m 次
{m,n}	{m,n}?	至少重复匹配前表达式 m 次，至多重复匹配前表达式 n 次

由表 2-20 和表 2-21 可知，“.*”为最大匹配，能匹配源字符串中所有能匹配的字符串。“.*?”为最小匹配，只匹配第一次出现的字符串。例如，d.*g 能匹配任意以 d 开头、以 g 结尾的字符串，如“debug”和“debugging”，甚至“dog is walking”都能匹配。而 d.*?g 只能匹配“debug”，在“dog is walking”字符串中，则只能匹配到“dog”。

如果要实现更为复杂的匹配，可以用组合运算符，如表 2-22 所示。

表 2-22　组合运算符

组	描　述
[...]	匹配集合内的字符，如[a-z]、[1-9]或[,./;']
[^...]	匹配除集合外的所有字符，相当于取反操作
A|B	匹配表达式 A 或 B，相当于 or 操作
(...)	表达式分组，每对括号为一组，如([a-b]+)([A-Z]+)([1-9]+)
\number	匹配在 number 表达式组内的文本

有一组特殊的字符序列，用来匹配具体的字符类型或字符环境。如\b 匹配字符边界，food\b 匹配“food”、“zoofood”，而与“foodies”不匹配。这些特殊的字符序列见表 2-23。

表 2-23　特殊字符序列

字　符	描　述
\A	只匹配字符串的开始
\b	匹配一个单词边界
\B	匹配一个单词的非边界
\d	匹配任意十进制数字字符，等价于 r‘[0-9]’
\D	匹配任意非十进制数字字符，等价于 r‘[^0-9]’
\s	匹配任意空白字符(空格符、Tab 制表符、换行符、回车、换页符、垂直线符号)
\S	匹配任意非空白字符
\w	匹配任意字母数字字符
\W	匹配任意非字母数字字符
\Z	仅匹配字符串的尾部
\\	匹配反斜线字符

在正则表达式中，还有一套声明(assertion)，可用于对具体事件进行声明，如表 2-24 所示。

表 2-24　正则表达式声明

声　明	描　述
(?iLmsux)	匹配空字符串，iLmsux 字符对应后文中的正则表达式修饰符
(?:...)	匹配圆括号内定义的表达式，但不填充字符组表
(?P)	匹配圆括号内定义的表达式，但匹配的表达式还可用作 name 标识的符号组
(?P=name)	匹配所有与前面命名的字符组相匹配的文本
(?#...)	引入注释，忽略圆括号中的内容
(?=...)	如果所提供的文本与下一个正则表达式元素匹配，这之间没有多余的文本就匹配。这允许在一个表达式中进行超前操作，而不影响正则表达式其余部分的分析。如“Martin”其后紧跟“Brown”，则“Martin(?=Brown)”就只与“Martin”匹配
(?!...)	仅当指定表达式与下一个正则表达式元素不匹配时匹配，是(?=...)的反操作
(?<=...)	如果字符串当前位置的前缀字符串是给定文本，就匹配，整个表达式就在当前位置终止。如(?<=abc)def 表达式与“abcdef”匹配。这种匹配是对前缀字符数量的精确匹配
(?<!...)	如果字符串当前位置的前缀字符串不是给定的正文，就匹配，是(?<=...)的反操作

正则表达式还支持一些修饰符(见表 2-25)，它会影响正则表达式的执行方法。

表 2-25　正则表达式的常用修饰符

修 饰 符	描　述
I	忽略表达式的大小写来匹配文本
L	根据当前语言环境解释单词。这种解释影响字母组(\w 和\W)以及字边界行为(\b 和\B)
M	多行匹配。就是匹配换行符两端的潜在匹配。影响正则中的^$符号
S	使一个句点(.)匹配任何字符，包括换行符
U	根据 Unicode 字符集解释字母。此标志影响\w、\W、\b、\B 的行为

2.8.2　正则表达式的操作举例

通过 Python 内建的 re 模块，我们就可以在 Python 中利用正则表达式对字符串进行搜索、抽取和替换操作了。例如，使用 re.search()函数能够执行一个基本的搜索操作，它能返回一个 MatchObject 对象；使用 re.findall()函数能够返回一个匹配列表。例如：

```
>>> import re
>>> a="this is my re module test"
>>> obj = re.search(r'.*is',a)
>>> print (obj)
<_sre.SRE_Match object; span=(0, 7), match='this is'>
>>> obj.group()
'this is'
>>> re.findall(r'.*is',a)
['this is']
>>>
```

搜索操作返回的 MatchObject 对象有很多方法可供使用，其功能如表 2-26 所示。

表 2-26　MatchObject 对象的常用方法

方　法	描　述
expand(template)	展开模板中用反斜线定义的内容
m.group([group,...])	返回匹配的文本，是个元组。此文本是与给定 group 或由其索引数字定义的组匹配的文本，如果没有给定组名，则返回所有匹配项
m.groups([default])	返回一个元组，该元组包含模式中与所有组匹配的文本。如果给出 default 参数，default 参数值就是与给定表达式不匹配的组的返回值。default 参数的默认取值为 None
m.groupdict([default])	返回一个字典，该字典包含匹配的所有子组。如果给出 default 参数，其值就是那些不匹配组的返回值。default 参数的默认取值为 None
m.start([group])	返回指定 group 的开始位置，或返回全部匹配的开始位置
m.end([group])	返回指定 group 的结束位置，或返回全部匹配的结束位置
m.span([group])	返回两元素组，此元组等价于关于一给定组或一个完整匹配表达式的(m.start(group), m.end(group)))列表
m.pos	传递给 match()或 search()函数的 pos 值
m.endpos	传递给 match()或 search()函数的 endpos 值

续表

方　法	描　述
m.re	创建这个 MatchObject 对象的正则式对象
m.string	提供给 match()或 search()函数的字符串

使用 sub()或 subn()函数，可以在字符串上执行替换操作。sub()函数的基本格式如下：

```
sub(pattern,replace,string[,count])
```

举例如下：

```
>>> str = 'The dog on my bed'
>>> rep = re.sub('dog','cat',str)
>>> print (rep)
The cat on my bed
>>>
```

其中的 replace 参数可接受函数。要获得替换的次数，可使用 subn()函数(基本格式同 sub()函数)，该函数返回一个元组，此元组包含替换了的文本及替换的次数。例如：

```
>>> str = 'The dog on my bed'
>>> rep = re.subn('dog','cat',str)
>>> print (rep)
('The cat on my bed', 1)
>>>
```

若需要用同一个正则式进行多次匹配操作，则可以把正则表达式编译成内部语言，这样可以提高处理速度。编译正则表达式使用 compile()函数，其基本格式为：

```
compile(str[, flags])
```

其中，str 表示需要编译的正则表达式串，flags 是一个修饰标志符。正则表达式被编译后生成一个对象，该对象拥有多种方法和属性，如表 2-27 所示。

表 2-27　正则表达式编译后对象的方法/属性

方法/属性	描　述
r.search(string[,pos[,endpos]])	同 search()函数，但此函数允许指定搜索的起点和终点
r.match(string[,pos[,endpos]])	同 match()函数，但此函数允许指定搜索的起点和终点
r.split(string[,max])	同 split()函数
r.findall(string)	同 findall()函数
r.sub(replace,string[,count])	同 sub()函数
r.subn(replace,string[,count])	同 subn()函数
r.flags	创建对象时定义的标志
r.groupindex	将 r '(?Pid)'定义的符号组名字映射为组序号的字典
r.pattern	在创建对象时使用的模式

有两个函数在此值得一提，一是 re.escape()函数，二是 re.getattr()函数。

1. re.escape()函数用于转义字符串

在使用 Python 的过程中，对转义字符的使用也有苦恼之时。因为有时候我们需要使用

一些特殊符号，如“$ * . ^”等的原意，有时候需要的是被转义后的功能，并且转义字符的使用很繁琐时，很容易出错，re.escape()是解决这一问题的灵丹妙药。

re.escape(pattern)可以对字符串中所有可能被解释为正则运算符的字符进行转义。如果字符串很长，且包含很多特殊字符，而又不想输入一大堆反斜杠，或者字符串来自于用户(比如通过 input 函数获取输入的内容)，且要用作正则表达式的一部分的时候，就可以使用这个函数。

现举一例说明之：

```
>>> re.escape('www.python.org')
'www\\.python\\.org'
>>> re.findall(re.escape('w.py'),"jw.pyji w.py.f")
['w.py', 'w.py']
>>> |
```

这里的 re.escape(‘w.py’)用作 re.findall()函数的正则表达式部分。

2. re.getattr()函数用于获取对象的引用

现举例说明如下：

```
>>> li=['a','b']
>>> getattr(li,'append')
<built-in method append of list object at 0x02508F58>
>>> getattr(li,'append')('c')          #相当于li.append('c')
>>> li
['a', 'b', 'c']
>>> handler=getattr(li,'append',None)
>>> handler
<built-in method append of list object at 0x02508F58>
>>> handler('cc')                      #相当于li.append('cc')
>>> li
['a', 'b', 'c', 'cc']
>>> result = handler('bb')
>>> li
['a', 'b', 'c', 'cc', 'bb']
>>> print (result)
None
>>> |
```

2.8.3　正则表达式测试工具

按照特定需求编写的正则表达式到底正确与否，初学者往往拿捏不准，因此，“正则表达式测试工具”应运而生，它为程序员编写正则表达式带来了极大的方便，不仅能帮助程序员编写需要的正则表达式，还可以使用它理解别人编写的表达式。

目前，正则表达式测试工具软件非常多，如 RegexBuddy、RegexMagic、Regex Match Tracer、RegexTester 等。因篇幅所限，本小节仅介绍一款免费的国产正则表达式工具软件 Regex Match Tracer，简称 Match Tracer。

Match Tracer 是一款用来编写和调试正则表达式的工具软件，通过其可视化的界面，我们可以快速、正确地写出复杂的正则表达式。

图 2-1 是 Match Tracer 启动后的界面。

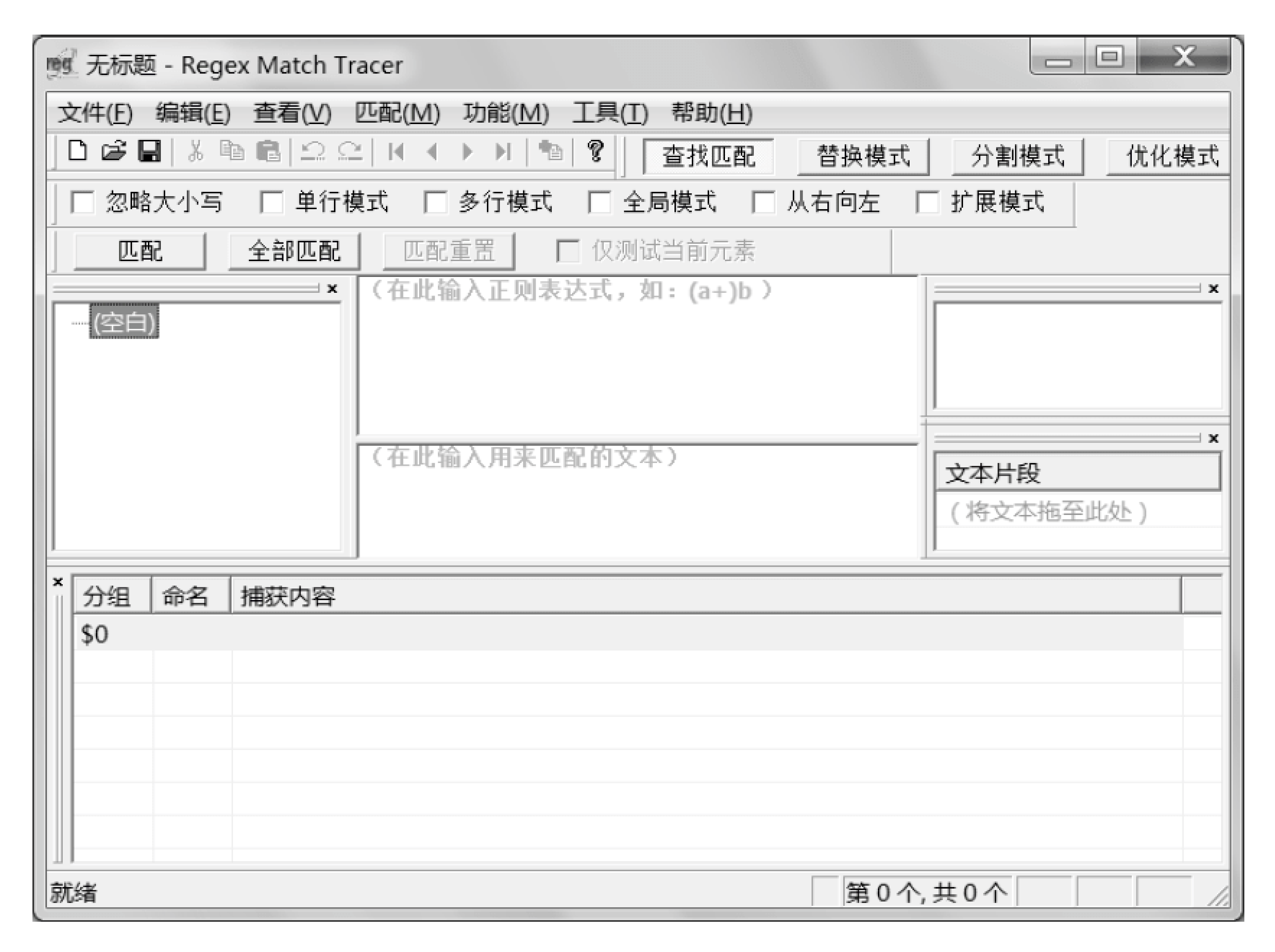


图 2-1 Match Tracer 界面

该软件的主要功能如下。

(1) 以语法着色形式显示表达式，使正则表达式便于阅读。

(2) 采用树和分组列表，同步显示正则表达式的结构，使复杂的表达式一目了然、长而不乱。

(3) 详细记录每一个匹配结果，包含分组结果以及匹配所花时间。

(4) 可进行“匹配”、“替换”、“分割”功能的正则表达式应用测试。

(5) 可进行“忽略大小写”、“单行模式”、“多行模式”、“全局模式”、“从左向右”、“扩展模式”等模式下的正则表达式测试。

(6) 可单独测试表达式中的一部分，有利于分段调试复杂的正则表达式。

(7) 可以设置一个匹配起始点，方便排查表达式错误。

(8) 支持高级正则语法，例如递归匹配等。

(9) 可以保存文本片段，例如表达式或者其他文本，也可以跟任意其他编辑器之间进行相互拖动。

(10) 可以将当前表达式保存为一个“快照”，使用户放心地改写表达式。

(11) 强调编写“复杂”的正则表达式。

一个完善的表达式往往都是比较复杂的，比如分析 HTML 的表达式。但是，复杂的表达式并不意味着低效，相反，因为复杂的表达式考虑得比较周全，所以匹配效率反而更高。Match Tracer 正是针对这种情况，着重考虑如何协助编写复杂而周全的表达式。

该软件可将测试好的表达式直接导出为程序语言代码，也可以直接从程序源代码的字符串中导入表达式；支持匹配结果、替换结果、分割结果的导出，整个表达式测试环境可以另存为一个项目。

下面详细介绍该软件的最常用功能。

1. 查找匹配

“查找匹配”界面适合用来进行一般的正则表达式匹配测试。如图 2-2 所示，界面中间有两个编辑框，上面的编辑框用于输入正则表达式，下面的编辑框用于显示匹配的文本；界面左侧中部是一个显示正则表达式结构的树结构控件；界面下方是一个列举正则表达式捕获组的列表框。

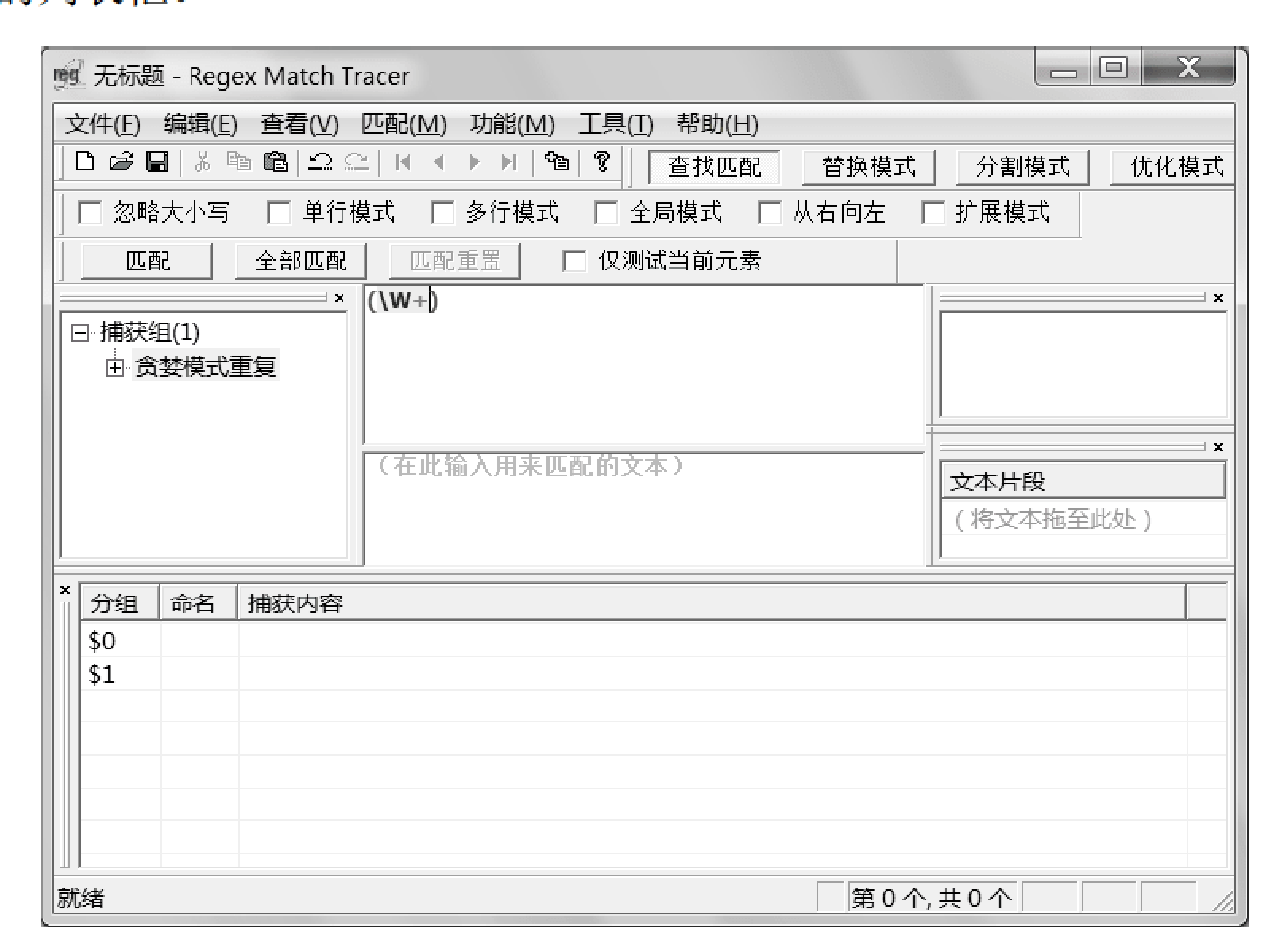


图 2-2　“查找匹配”界面

(1)　正则表达式输入框。

在正则表达式输入框内输入表达式时，表达式文本将根据表达式的语法自动进行着色。随着正则表达式的输入，正则表达式结构框和捕获组列表框会同步显示。当输入光标在编辑框中移动时，光标所在位置的当前表达式元素会突出显示。在表达式输入框中双击鼠标，鼠标所在位置的当前元素会被选中。

(2)　匹配文本编辑框。

编辑文本，用于测试正则表达式。

(3)　表达式树结构框。

在表达式输入框中输入表达式时，树结构会同步更新。当输入光标在输入框中移动时，树结构中相应的节点会被选中。点击树结构中的节点，表达式框中相应的元素会被选中。双击树结构框，表达式输入框将会获得输入焦点。

(4)　捕获组列表框。

在输入框中输入表达式时，捕获组框会同步更新，列举当前表达式中的捕获组。点击捕获组列表框，匹配文本编辑框中的相应文本会被选中。双击捕获组列表框，表达式输入框将会获得输入焦点，如图 2-3 所示。

2. 替换模式

“替换模式”与“匹配模式”相比，增加了“替换为”输入框和替换结果框，如图 2-4

所示。

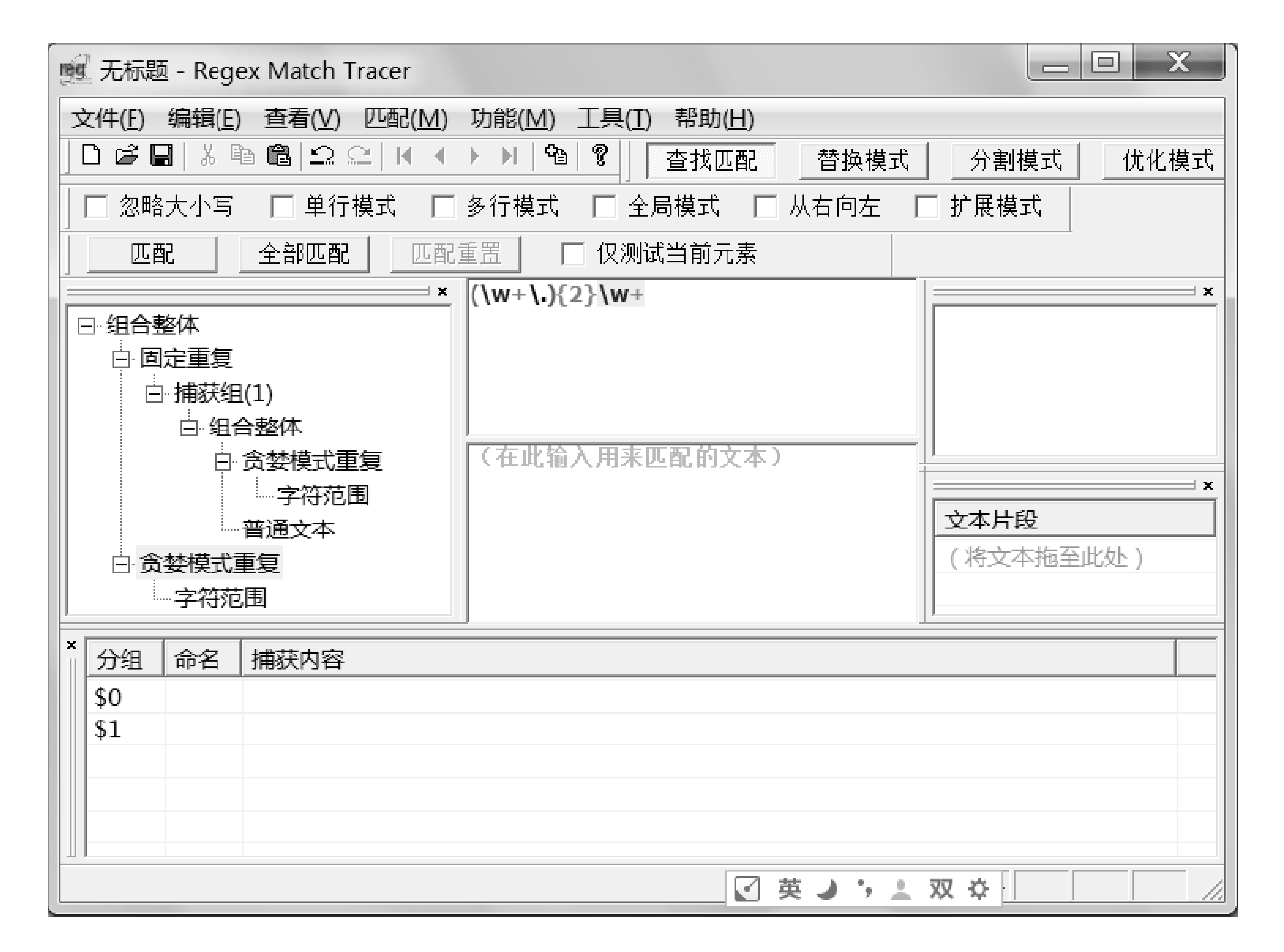


图 2-3 双击捕获组列表框

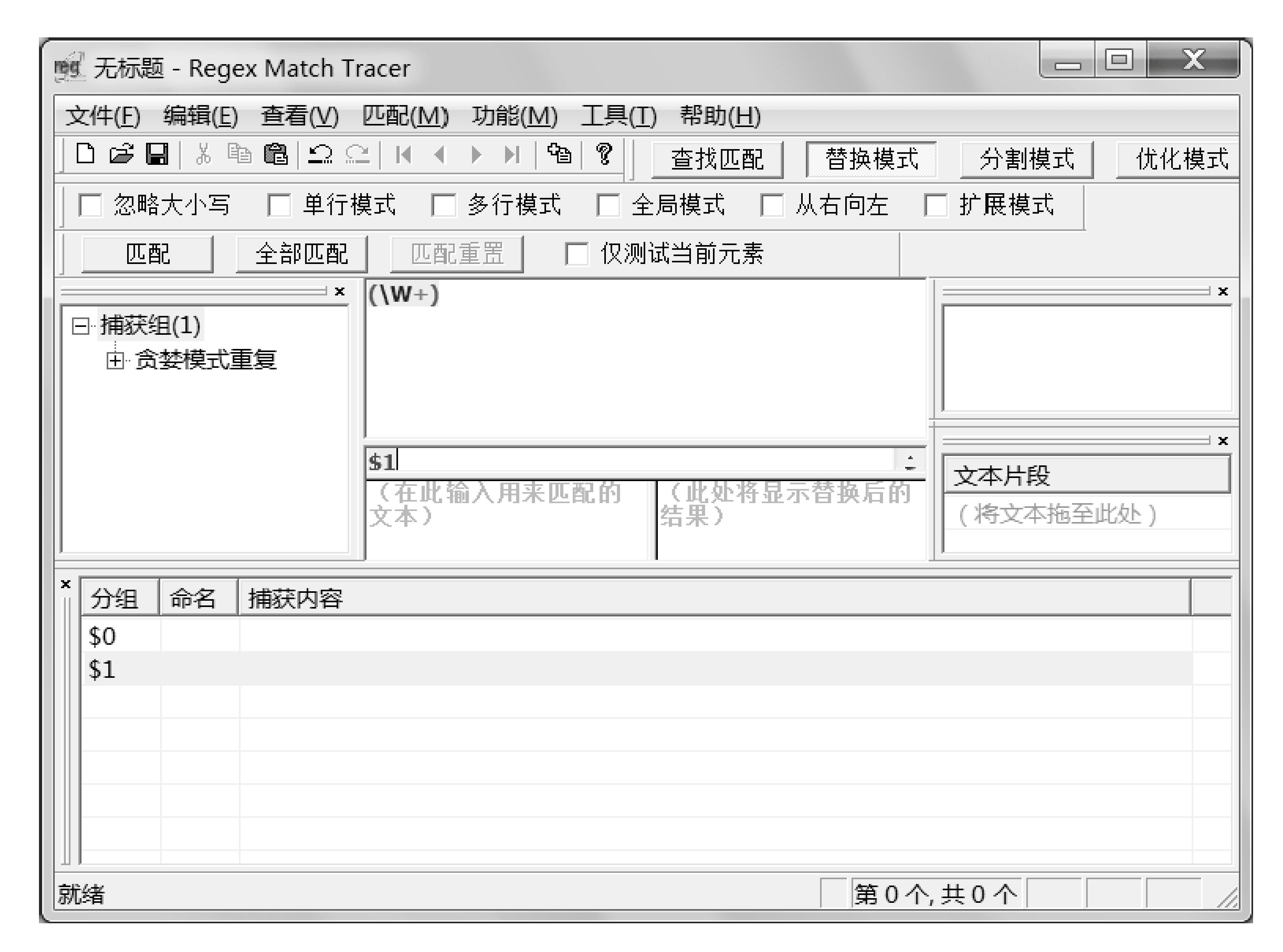


图 2-4 “替换模式”界面

图中写着“$1”字样的编辑框为“替换为”输入框。再往下的两个分隔开的编辑框中，左边一个是匹配文本输入框，右边一个是替换结果框。替换结果框为只读编辑框，因为是替换的结果，因此其中的内容不可再次编辑。

(1) “替换为”输入框。

在“替换为”输入框中输入时，以‘$’开始的特殊符号会采用突出颜色显示。输入光标在“替换为”框中移动时，匹配文本框和结果框中对应的文本会被选中。

(2) 替换结果框。

点击“匹配”按钮或者“全部匹配”按钮时，替换结果会显示在替换结果框。输入光标在替换结果框中移动时，匹配文本框和“替换为”输入框中对应的文本会被选中，实际运行效果如图 2-5 所示。可见，待匹配文本中的网址都被统一替换为目标字段。

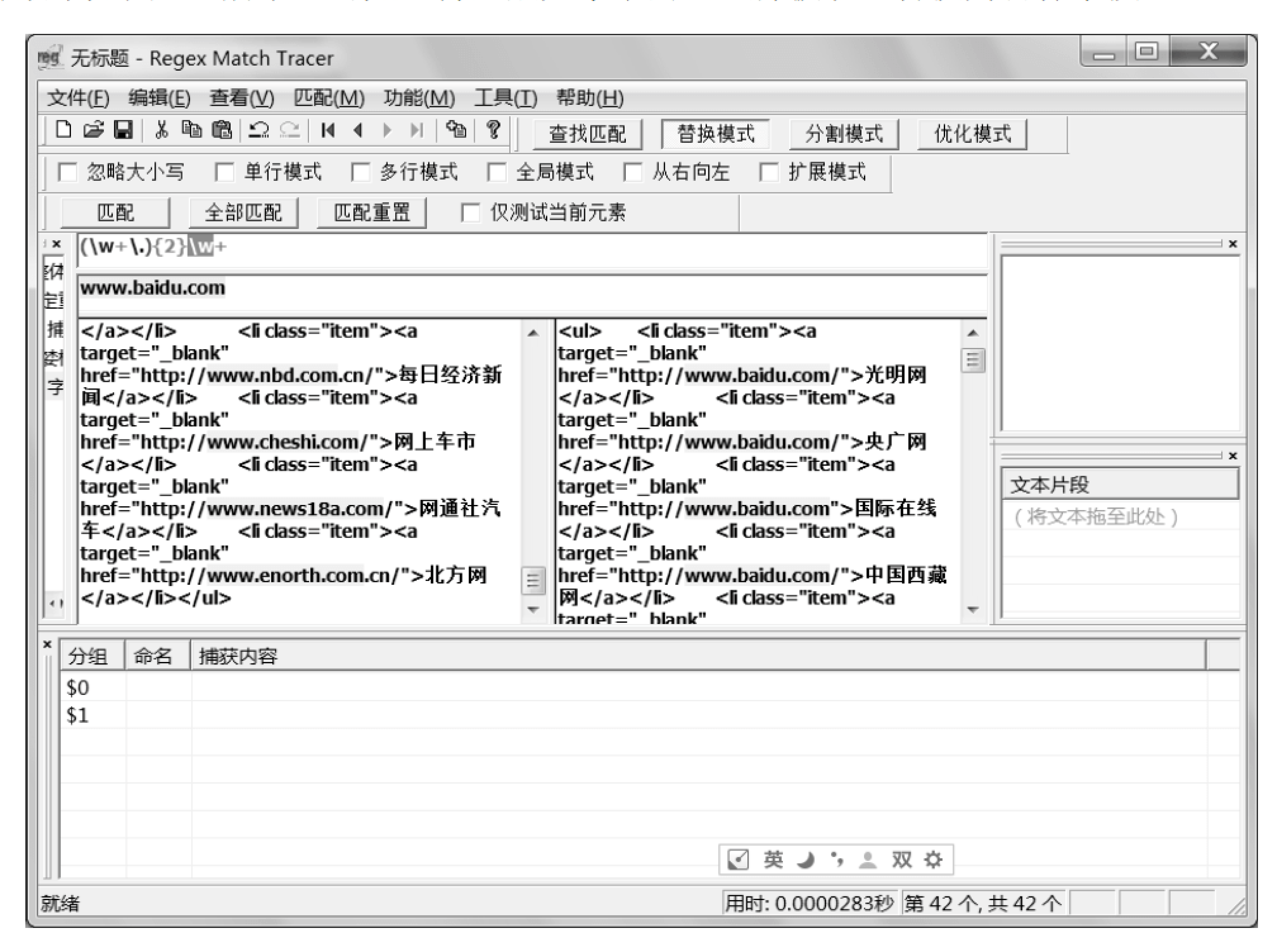


图 2-5　替换模式实际运行效果

3. 文件菜单

文件菜单负责“正则表达式项目”的新建、打开和保存。正在进行编辑和测试的正则表达式、匹配文本以及相关的其他参数，可以一起作为一个“正则表达式项目”保存到一个文件中，下次打开后可继续进行编辑和测试。

2.8.4　正则表达式的在线测试

如果读者在实际写正则表达式的时候，对匹配范围掌控不好，可以在正则表达式测试网站 http://tool.chinaz.com/regex/上修改正则匹配规则，通过这个正则测试网站，可以清晰地看到匹配的内容。

此外，对于一些常用的正则表达式，如中文字符、双字节字符、空白行、E-mail 地址、网址、URL、手机号码(国内)、电话号码(国内)、正浮点数、负浮点数、整数、腾讯 QQ 号、邮政编码、IP、身份证号、格式日期、正整数、负整数、用户名等，该网站均直接给出，大大方便了使用者。

本 章 小 结

作为 Python 语言基础中的基础，本章介绍了 Python 的基本语法与句法，包括标识符、注释符、换行符、续行符等，介绍了代码块的概念及其表达方式，强调了 Python 文件的模块化组织方式；介绍了 Python 语言的基本数据类型，包括数值、字符串、列表、元组、字典、集合；介绍了赋值的概念，介绍了序列的概念及其基本操作，包括索引、切片、加、乘、检查成员等；介绍了字符串、列表、元组、字典、集合中大量的函数(方法)；最后介绍了正则表达式的概念及应用。

习　　题

1. 填空题

(1) Python 的注释语句以(　　)字符开始。

(2) Python 的相邻语句使用换行(回车)分隔，亦即一行一条语句。如果一行语句过长，可以使用续行符(　　)分解为多行。

(3) Python 允许将多个语句写在同一行上，语句之间用(　　)隔开。

(4) Python 中代码组以不同的(　　)分隔。

(5) Python 支持复数，可以表示为 a+bj 或者(　　)。

(6) 运算符的优先级决定了运算顺序，例如，2*2**3 的结果是(　　)，2**3/2 的结果是(　　)。

(7) 设有 s1 = set('hello'), s2 = set('world')，则 s1 & s2 的结果是(　　)，s1 | s2 的结果是(　　)，s1 - s2 的结果是(　　)，s1 ^ s2 的结果是(　　)。

(8) 设有字符串 str1 = 'Hello ', str2 = 'world'，则 str1+str2 的结果是(　　)，str1*2 的结果是(　　)。

(9) 当在多个正则表达式方法中使用同一匹配模式时，可以通过(　　)函数将匹配模式编译成内部语言，以提高处理速度。

(10) 设字典 dict1 = {'addr':'天津', 'name':'Bob', 'addr':'北京'}，则语句 print(dict1['addr'])的结果是(　　)。

2. 选择题

(1) 哪一个不是 Python 语言的基本结构？(　　)

A. 链表　　B. 元组　　C. 字典　　D. 集合

(2) 那一个不属于 Python 的序列类型？(　　)

A. 字符串　　B. 列表　　C. 集合　　D. 元组

(3) /和//都是 Python 的除法运算，6/3 的结果是(　　)，6//3 的结果是(　　)。

A. 2　　B. 3　　C. 2.0　　D. 0

(4) 以下哪个变量的命名不正确？(　　)

A. mm_123　　B. _mm123_　　C. 123_mm　　D. _mm_123

(5) type(1+2*3.14)的结果是(　　)。

A. <type ‘int’>　　B. <type ‘float’>

C. <type ‘str’>　　D. <type ‘complex’>

(6) 有字符串 str=‘abcdefg’，那么 str[:-4:-2]的结果是(　　)。

A. ‘ac’　　B. ‘ge’　　C. ‘ca’　　D. ‘eg’

(7) 执行下面的操作后，list2 的值是(　　)。

```
list1 = [4,5,6]
list2 = list1
list1[2] = 3
```

A. [4,5,6]　　B. [4,3,6]　　C. [4,5,3]　　D. 以上都不正确

(8) 关于 Python 中的变量，下列说法错误的是(　　)。

A. 变量不必事先声明　　B. 变量须先定义后使用

C. 对象无须指定类型　　D. 可以使用 del 释放变量

(9) 哪一个数据类型不可以作为字典的键？(　　)

A. 字符串　　B. 数字　　C. 元组　　D. 列表

(10) 下列哪种不属于集合的操作？(　　)

A. &　　B. |　　C. +　　D. -

3. 问答题

(1) 表中 L 有相同的元素，如何用最简单的方法去除之？

(2) 以下列三种方式打印字符串<Bob said “I’ m OK”>。

① print(r‘Bob said “I\’m OK”’)

② print(r‘Bob said “I’ m OK”’)

③ print(‘Bob said “I\’ m OK”’)

问：哪种打印方式是正确的？不正确的打印方式的错误原因是什么？

(3) 对元组 tup = (1,2,[3,4],‘567’)进行以下操作。

① tup[0] = 8

② tup[2][0] = 8

③ tup[3][0] = 8

④ tup[2] = [8,9]

问：上述哪些操作是正确的，请说明理由。

(4) 有如下字符串定义：

```
str1 = 'abc'
str2 = str1.replace('a', 'A')
```

问：id(str1)是否等于 id(str2)？请说明理由。

4. 实验操作题

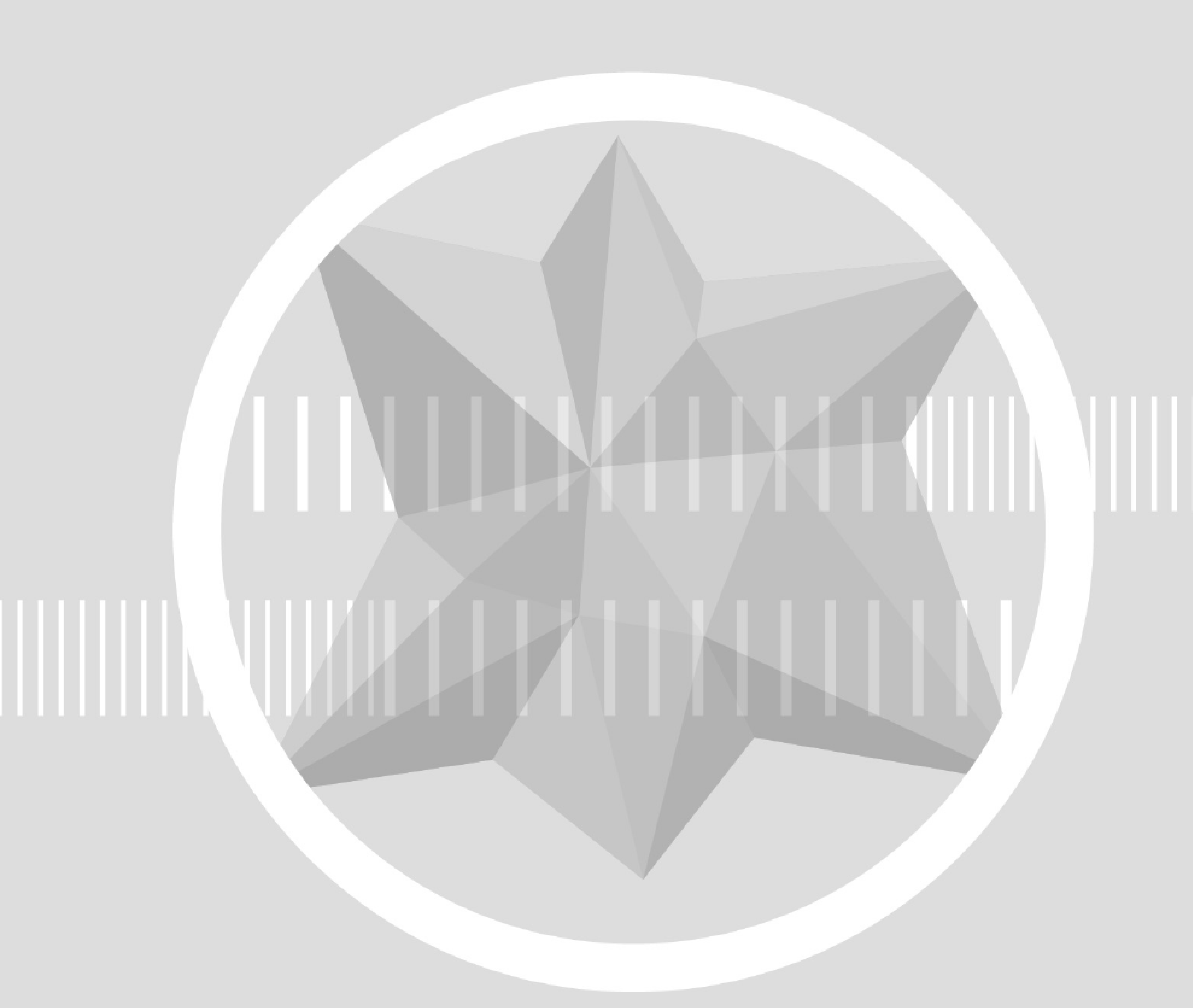
(1) 列表 names = [‘Dave’,[‘Mark’,‘Ann’],‘Phil’,‘Tom’]，根据以下要求，写出相应的列表操作。

① 获取元素‘Mark’。
② 将元素‘Phil’换成‘Jeff’。
③ 获取列表的长度。
④ 向列表末尾添加‘Kate’元素。
⑤ 移除列表中的‘Ann’元素。
⑥ 将‘Sydney’插入到列表下标为 2 的位置。
⑦ 分别获取列表的前两个元素和后两个元素。

(2) 写出下列各条命令的执行结果:

```
info = {"stu01":"张三","stu02":"李四","stu03":"王五"}
print(info)
print(info["stu01"])
print(info.get("stu04"))
print("stu03" in info)
info["stu02"] = "李四四"
print(info)
info["stu04"] = "赵六"
print(info)
del info["stu04"]
info.pop("stu03")
print(info)
```

(3) 根据要求写出相应的正则表达式。
① 匹配正整数。
② 匹配负整数。
③ 匹配整数。
④ 匹配由 26 个大写英文字母组成的字符串。
⑤ 匹配由 26 个小写英文字母组成的字符串。
⑥ 匹配由 26 个英文字母组成的字符串。
⑦ 匹配由数字和 26 个英文字母组成的字符串。
⑧ 匹配中国邮政编码。
⑨ 匹配身份证(15 位或 18 位)。



第 3 章

Python 流程控制

本章要点

(1) if 语句的基本格式及执行规则。
(2) if 语句嵌套的使用方法。
(3) for 语句的基本格式及执行规则。
(4) for 语句循环嵌套的使用方法。
(5) range()函数在循环中的使用。
(6) while 语句的基本格式及执行规则。
(7) while 语句循环嵌套的使用方法。
(8) break、continue、pass 等关键字在循环中的使用方法。

学习目标

(1) 理解选择结构和循环结构的执行过程。
(2) 掌握选择结构和循环结构的编程方法。
(3) 理解循环嵌套的执行过程。
(4) 运用流程控制编程解决一些实际问题。

本章将详细介绍在 Python 中如何实现选择结构，即根据特定的条件执行或不执行某些语句。本章还将介绍循环结构，即根据特定的条件多次执行某些语句。本章的诸多示例程序表明，程序中只有使用选择结构和循环结构，才能充分挖掘和利用计算机的功能，从而编写程序完成各种各样的任务。

3.1 if 语句

if 语句的基本内容包括 if、if-else、if-elif-else、三元运算符、比较运算符等。

Python 3 的条件语句通过测试某个条件是否成立(True 或者 False、真值或者假值、0 或者 1)来决定执行的代码块。if 后面的条件可以是一个，也可以是多个。多个条件通过布尔操作符 and(与)、or(或)、not(非)进行组合，形成条件表达式。

单个条件 if 语句的执行过程如图 3-1 所示。

3.1.1 if 语句

Python 中，if 语句的一般形式为：

```
if condition 1:
   statements_1
```

其执行过程是：如果 condition_1 为 True，则执行 statements_1 语句块(代码块)。

该 if 语句由三部分组成：关键字、判断结果真假的条件表达式，以及当表达式为真时执行的代码块。

代码块是条件为真时执行的一组代码，在代码前放置空格来缩进语句，即可创建代码块。前已提及，Python 要求严格缩进，不能第一次使用 Tab 键缩进，第二次使用空格键缩

进，而要做到缩进方式的统一。另外，在集成开发环境中，缩进是自动完成的，换句话说，键入冒号(:)并按 Enter 键后，下一行一般将自动缩进四个空格。

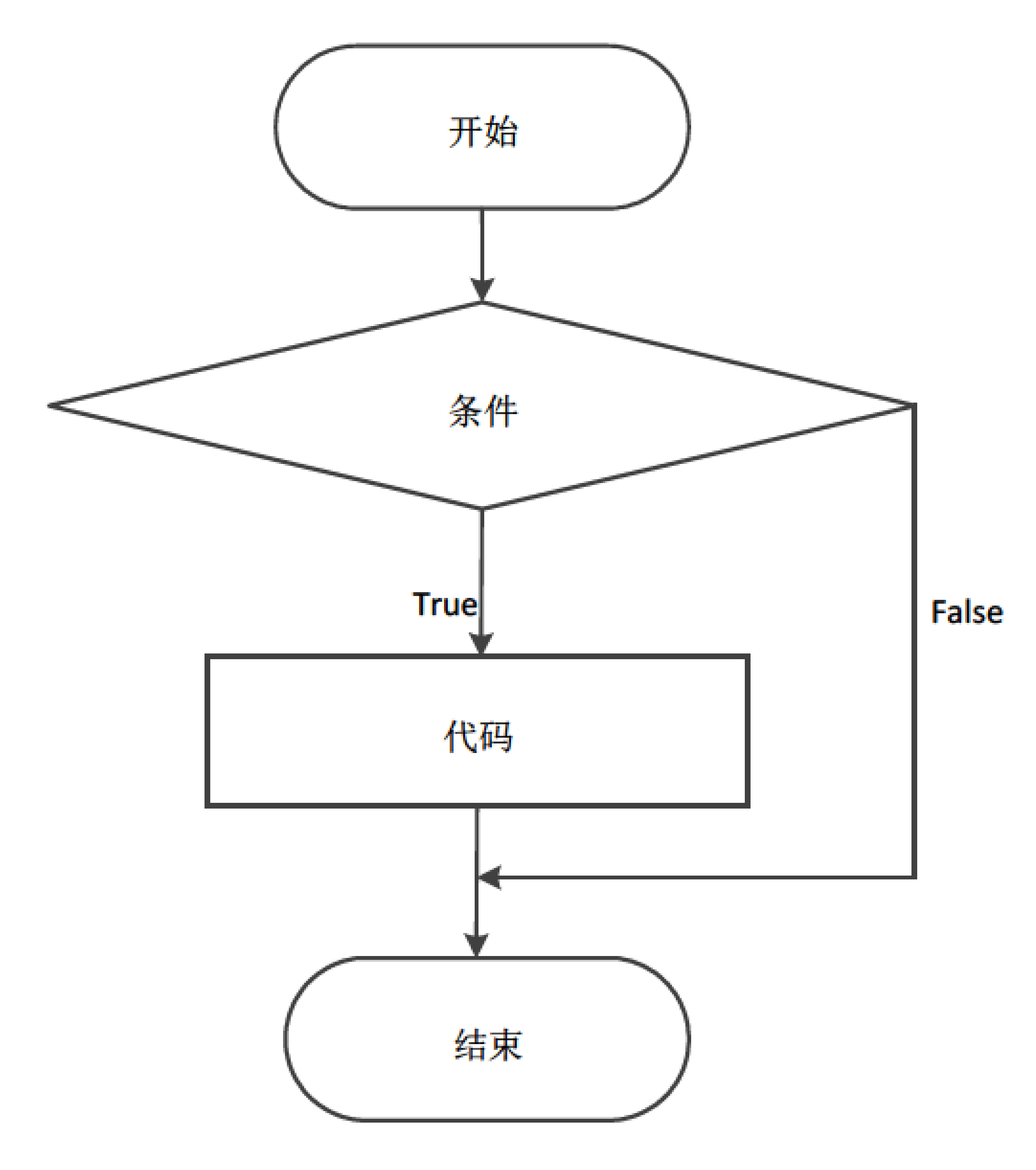


图 3-1　if 语句的控制流程

【例 3-1】简单的 if 语句：

```
var1 = 50
if var1:
   print ("if-1 表达式条件为 true 或 1")
   print (var1)

var2 = 0
if var2:
   print ("if-2 表达式条件为 true 或 1")
   print (var2)
print ("Good bye!")
```

执行以上代码，输出结果为：

```
if-1表达式条件为true或1
50
Good bye!
>>>
```

从结果可以看到，由于变量 var2 为 0，所以对应的 if 内的语句块没有执行。

注意：

- 在每个条件表达式的后面都要使用冒号(:)，表示接下来是满足条件后要执行的语句块。
- 在 Python 中，没有类似于 C 语言的 switch-case 语句。
- 标准值 False 和 None、所有类型的数字 0(浮点型、长整型和其他类型)、空序列(空字符串、空的元组和列表及字典等)都为假或者 0。

- if 语句块如果仅有一条语句，则可以与 if 语句写在同一行上，但 “:” 不能省略。

3.1.2 if-else 语句

Python 中，if-else 语句的一般形式如下：

```
if condition 1:
    statements_1
else:
    statements_2
```

其执行过程是：

如果 condition_1 为 True，则执行 statements_1 语句块。

如果 condition_1 为 False，则执行 statements_2 语句块。

其中，else 子句是可选的。

【例 3-2】根据输入的年龄判断是否为成年人：

```
age = int(input("请输入你的周岁年龄: "))
print("")
if age < 18:
    print("你是未成年人")
else:
    print("你是成年人")
```

执行以上代码，运行程序两次，输出结果为：

```
请输入你的周岁年龄: 17

你是未成年人
>>> ================
>>>
请输入你的周岁年龄: 18

你是成年人
>>>
```

注意： input()函数用于从键盘接收输入的内容，int()函数用于把输入的内容转换为整数。这些都是内建函数，可以直接使用，不需要使用第 4 章要介绍的 import 语句导入相应模块。

3.1.3 if-elif-else 语句

Python 中，if-elif-else 语句的一般形式如下：

```
if condition 1:
    statements_1
elif condition 2:
    statements 2
else:
    statements_3
```

其执行过程是：

如果 condition_1 为 True，则执行 statements_1 语句块。

如果 condition_1 为 False，则判断 condition_2。

如果 condition_2 为 True，则执行 statements_2 语句块。

如果 condition_2 为 False，则执行 statements_3 语句块。

Python 中用 elif 代替了 else if，所以 if 语句的关键字为 if-elif-else。跟 else 一样，elif 子句是可选的，然而不同的是，if 语句可以有任意数量的 elif 子句，但最多只能有一个 else 子句。

【例 3-3】根据动物的年龄，计算对应于人类的年龄：

```
age = int(input("请输入狗的年龄： "))
print("")
if age < 0:
    print("请输入正确的年龄!")
elif age == 1:
    print("相当于 14 岁的人。")
elif age == 2:
    print("相当于 22 岁的人。")
elif age > 2:
    human = 22 + (age -2)*5
    print("对应人的年龄： ", human)
```

执行以上代码，输出结果为：

```
请输入狗的年龄： 8

对应人的年龄：  52
>>>
```

3.1.4　三元运算符

Python 三元运算符的语法为：

```
X if C else Y
```

其中 C 是条件表达式，X 是 C 为 True 时的结果，Y 是 C 为 False 时的结果。有了三元运算符后，只需要一行，就可以完成条件判断和赋值操作。

【例 3-4】三元运算符的应用：

```
x, y = 8, 7
bigger = x if x > y else y
print(bigger)
```

执行以上代码，输出结果为：

```
8
>>>
```

也可以用正常的 if-else 语句完成上述功能，代码如下：

```
x, y = 8, 7
if x > y:
```

```
    bigger = x
else:
    bigger = y
print(bigger)
```

可见，使用三元运算符能让程序更加简洁、紧凑。

3.1.5　比较操作符

第 2 章介绍运算符时，已在表 2-1 中罗列了 Python 的比较操作符。为方便起见，本章单独将比较操作符列于表 3-1 中。

表 3-1　比较操作符及其意义

比较操作符	比较操作符的含义		
<	小于	例：2<3　值为 True	3<1　值为 False
<=	小于或等于	例：4<=4　值为 True	4<=1　值为 False
>	大于	例：9>7　值为 True	7>8　值为 False
>=	大于或等于	例：9>=9　值为 True	7>=8　值为 False
==	等于，比较对象是否相等	例：5==5　值为 True	4==8　值为 False
!=	不等于	例：5!=4　值为 True	4!=4　值为 False
is	a is b	a　和 b　是同一个对象	
is not	a is not b	a　和 b 不是同一对象	
in	a in b	a　是 b 容器中的元素	
not in	a not in b	a 不是　b 容器中的元素	

下面我们看几个例子。

【例 3-5】“==”操作符的功能：

```
#使用常量
print(5 == 7)
#使用变量
x = 5
y = 5
print(x == y)
```

执行以上代码，输出结果为：

```
False
True
>>>
```

【例 3-6】演示猜数字功能：

```
x = int(input("请输入数字: "))
if x < 0:
    print('输入的数字小于 0')
elif x == 0:
    print('输入的数字为 0')
elif x == 1:
```

```
    print('输入的数字为 1')
else:
print('输入的数字大于等于 2')
```

执行以上代码 4 次，输出结果分别为：

```
请输入数字: 5
输入的数字大于等于2
>>> =============
>>>
请输入数字: -1
输入的数字小于0
>>> =============
>>>
请输入数字: 1
输入的数字为1
>>> =============
>>>
请输入数字: 0
输入的数字为0
>>> |
```

【例 3-7】判断列表是否相等：

```
a = c = [1,2]
b =[1,2]
print(a==c)
print(a==b)
print (a is c)
print (a is b)
```

执行以上代码，输出结果为：

```
True
True
True
False
>>>
```

注意： 此程序说明 a 和 b 值相等，但不是同一个对象。

【例 3-8】字符串应用：

```
x = "hello"
if 'e' in x:
   print("yes")
else:
   print("no")
```

执行以上代码，输出结果为：

```
yes
>>>
```

3.1.6　if 嵌套

在嵌套的 if 语句中，可以把 if-elif-else 结构放在另外一个 if-elif-else 结构中。一般形式如下：

```
if condition 1:
    statements 1
    if condition 2:
        statements 2
    elif condition 3:
        statements 3
    else
        statements 4
elif condition 4:
    statements 5
else:
    statements_6
```

【例 3-9】判断输入的数字是否可以被 3 或 5 整除：

```
number=int(input("输入一个数字："))
if number%3==0:
    if number%5==0:
        print ("你输入的数字可以整除 3 或 5")
    else:
        print ("你输入的数字可以整除 3，但不能整除 5")
else:
    if number%5==0:
        print ("你输入的数字可以整除 5，但不能整除 3")
    else:
        print ("你输入的数字不能整除 3 或 5")
```

执行以上代码，运行 4 次，输出结果为：

```
输入一个数字：15
你输入的数字可以整除 3 或5
>>> ============================
>>>
输入一个数字：9
你输入的数字可以整除 3, 但不能整除 5
>>> ============================
>>>
输入一个数字：20
你输入的数字可以整除 5, 但不能整除 3
>>> ============================
>>>
输入一个数字：23
你输入的数字不能整除 3 或 5
>>>
```

3.2 for 循环

本节将介绍 Python 循环语句的使用。Python 中的循环语句有 for 和 while 两种，本节介绍 for 循环。

Python 中 for 循环的控制流程如图 3-2 所示。

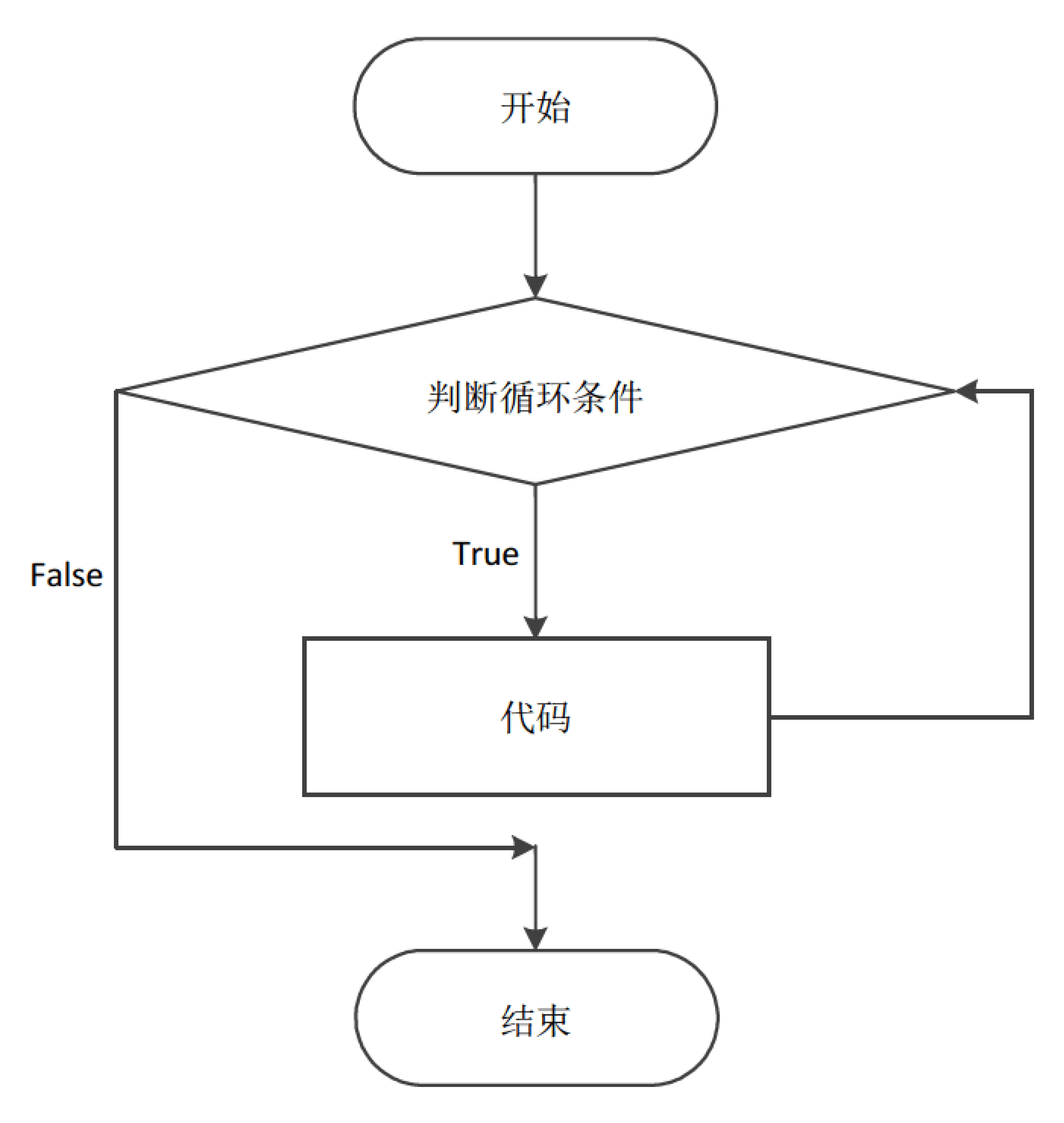


图 3-2　for 循环的控制流程

3.2.1　for 循环的基本结构

Python 中，for 循环可以遍历任何序列的项目，如字符串、元组、列表、文件等，也可以遍历非序列的字典。

for 循环的一般格式如下：

```
for <variable> in <sequence>:
    <statements 1>
else:
    <statements_2>
```

【例 3-10】 遍历字符串：

```
word = input("输入字符串：")
tripleWord = ""
for w in word:
    tripleWord += w*3
print("三倍字符串是：" + tripleWord + "。")
```

执行以上代码，输出结果为：

```
输入字符串：aeiou
三倍字符串是：aaaeeeiiiooouuu。
>>>
```

【例 3-11】 遍历元组：

```
languages = ("C", "C++", "JAVA", "Python")
```

```
for x in languages:
    print (x)
```

执行以上代码，输出结果为：

```
C
C++
JAVA
Python
>>>
```

【例 3-12】遍历列表：

```
people = ["美国人","英国人","德国人","法国人","韩国人","俄国人","Chinese",]
for i in range(len(people)):
    people[i] = people[i][0:2]
print(people)
```

执行以上代码，输出结果为：

```
['美国', '英国', '德国', '法国', '韩国', '俄国', 'Ch']
>>>
```

注意：

- range(len(people))相当于 range(7)，for 循环的 i 变量取值从 0 到 6，即：i＝0，1，2，3，4，5，6。
- people[i][0:2]是取每个 people 列表元素的前两个字符。

【例 3-13】遍历字典：

```
months = {'January':1, 'February':2, 'March':3, 'April':4, 'May':5}
for key in months:
    print(key,'--',months[key],end='  ')
    #end 保证同行输出并隔 2 个空格(详见 3.4.6 节)
print()                          #换行
for value in months.values():           #使用字典方法 values()
    print('values = ',value,end='  ')
print()
for key, value in months.items():        #使用字典方法 items()
    print(key,'--',value,end='  ')
```

执行以上代码，输出结果为：

```
January -- 1  February -- 2  April -- 4  March -- 3  May -- 5
values =  1  values =  2  values =  4  values =  3  values =  5
January -- 1  February -- 2  April -- 4  March -- 3  May -- 5
>>>
```

注意： 字典元素的输出顺序通常不固定。换句话说，迭代的时候，字典中的键和值都能保证被处理，但是处理顺序不确定。如果顺序很重要，可以将键值保存在单独的列表中或者排序输出。

【例 3-14】文本文件的行遍历：

```
firstName = input("输入姓氏：")
file = open("xingming.txt",'r')
for f in file:
```

```
    if f.startswith(firstName):
        print (f.rstrip())
file.close()
```

执行以上代码，输出结果为：

```
输入姓氏：张
张伯
张仲
张叔
张季
>>> |
```

注意：

- open()函数用于打开文件，即建立程序和文件的连接，允许程序从文件中读取数据。我们把程序和文件放到同一个文件夹中，这样就可以只用文件名，而不用文件的完整路径。
- xingming.txt 中有 6 个名字，分别是张伯、张仲、张叔、张季、李刚、王明，文本文件的每一行由一个换行符结束，rstrip()函数用于移除这个字符。
- close()函数用于关闭文件，即断开程序和文件的连接。
- open()和 close()两个函数将在第 5 章做具体介绍。

3.2.2 for 循环嵌套

Python 语言允许在一个循环体内嵌入另一个循环，其一般形式如下：

```
for <variable1> in <sequence1>:
    for <variable2> in <sequence2>:
        <statements 1>
    <statements_2>
```

【例 3-15】双重循环的应用：

```
for i in range(5,10):
    for j in range(5,10):
        print (i,"*",j,"=",i*j,end='')
        print("\t",end='')
```

执行以上代码，输出结果为：

```
5 * 5 = 25      5 * 6 = 30      5 * 7 = 35      5 * 8 = 40      5 * 9 = 45
6 * 5 = 30      6 * 6 = 36      6 * 7 = 42      6 * 8 = 48      6 * 9 = 54
7 * 5 = 35      7 * 6 = 42      7 * 7 = 49      7 * 8 = 56      7 * 9 = 63
8 * 5 = 40      8 * 6 = 48      8 * 7 = 56      8 * 8 = 64      8 * 9 = 72
9 * 5 = 45      9 * 6 = 54      9 * 7 = 63      9 * 8 = 72      9 * 9 = 81

>>>
```

第 1 行设定变量 i 从 5 到 10(不含 10)变化，每次自动加 1。

第 2 行设定变量 j 从 5 到 10(不含 10)变换，每次自动加 1。

第 3 行是打印语句，因为 print()函数的最后有 end=''(两个单引号，中间无空格，不是一个双引号，也可以是两个双引号，见下例)，所以每一次循环都不会换行，详细用法请参考 3.4.6 节。

第 4 行是表示输出一个制表符之后不换行。

注意：

- 双重循环的缩进格式很重要，这里再次强调，Python 有非常严格的语法规定。
- for 语句的判断标准是先判断后执行，所以 i=10 是不执行的，因此该程序是从 5×5=25 算到 9×9=81。

【例 3-16】双重循环输出举例：

```
number = int(input("输入一个数字："))
for i in range(0,number):
   for j in range(0,i+1):
      print("*",end="")
   print()
```

执行以上代码，输出结果为：

```
输入一个数字：4
*
**
***
****
>>>
```

3.2.3 for 循环中使用 else 分支

Python 的 for 循环有一个可选的 else 分支(类似 if 语句那样)，在循环迭代正常完成之后执行。换句话说，如果我们以正常的方式退出循环，那么 else 分支将被执行；但如果在循环体内执行了 break 语句、return 语句而退出循环，或者有异常出现而退出循环，那么 else 分支将不会被执行。下面考虑一个简单的例子。

【例 3-17】执行 else 分支的 for 循环：

```
for i in range(5):
   print(i)
else:
   print("没有更多的数字")
```

执行以上代码，输出结果为：

```
0
1
2
3
4
没有更多的数字
>>>
```

循环正常完成，所以 else 分支也被执行，并打印“没有更多的数字”。

如果循环所迭代的序列是空的，else 分支依然会被执行。

【例 3-18】迭代序列为空时执行 else 分支：

```
for i in {}:
  print(i)
else:
  print("for 循环后执行该语句")
```

执行以上代码，输出结果为：

```
for循环后执行该语句
>>>
```

如果用 break 语句终止循环，那么 else 分支将不会被执行。

【例 3-19】不执行 else 分支的 for 循环：

```
sites = ["Baidu", "Google","Lenovo","Apple"]
for site in sites:
    if site == "Lenovo":
        print("联想集团!")
        break
else:
    print("没有匹配数据!")
print("完成循环!")
```

执行以上代码，输出结果为：

```
联想集团!
完成循环!
>>>
```

注意：执行程序时，在循环到 Lenovo 时会跳出循环体，else 分支不再执行。

break 语句用于跳出循环体，具体用法参见 3.4.4 节。

3.2.4　列表解析

如果 list1 是一个列表，那么：

```
[f(x) for x in list1]
```

将创建一个新列表，并将 list1 中的每个元素放入新列表中，其中 f(x)为 Python 的内建函数或用户自定义函数。

【例 3-20】列表解析举例：

```
list1 = [2.3, 3.4, 4.5, 5.6, 6.7]
print ([int(x) for x in list1])        #将 list1 中的所有数转换为整数(割尾取整)
print ([int(x) for x in [2.3,3.4,4.5,5.6,6.7]]) #将列表中所有的数转换为整数
print ([int(x)**2 for x in list1])              #打印 list1 中所有数的平方
print ([int(x)**2 for x in list1 if int(x) % 2 == 0])
 #只打印 list1 中偶数的平方
```

执行以上代码，输出结果为：

```
[2, 3, 4, 5, 6]
[2, 3, 4, 5, 6]
[4, 9, 16, 25, 36]
[4, 16, 36]
>>>
```

注意：除列表外，列表解析也可以应用在其他对象上，如字符串、元组和用 range() 函数产生的算术表达式。

【例 3-21】应用在其他对象上的列表解析：

```
print ([ord(x) for x in "abcd"])
print ([int(x)**0.5 for x in (-1,9,16) if x >= 0])
print ([x**3 for x in range(3)])
```

执行以上代码，输出结果为：

```
[97, 98, 99, 100]
[3.0, 4.0]
[0, 1, 8]
>>>
```

3.3　range()函数

range()函数在前面已使用多次，读者对这一内建函数的用法已略知一二。由于其应用颇广，本节对此单独做详细介绍。

函数原型：range(start,end,step)

参数含义：

start——计数从 start 开始，默认是从 0 开始。例如 range(5)等价于 range(0,5)。

end——计数到 end 结束，但不包括 end。例如 range(0,5)是[0, 1, 2, 3, 4]，没有 5。

step——每次跳跃的间距，或称步长，默认为 1。例如 range(0,5)等价于 range(0, 5, 1)。

例如：

range(3,12,2)产生的序列是 3、5、7、9、11。

range(0,22,4)产生的序列是 0、4、8、12、16、20。

range(-10,20,5)产生的序列是-10、-5、0、5、10、15(注意没有 20)。

如果步长值为负数，并且初始值大于终止值，则 range()函数产生一个递减的序列，它由初始值开始，递减至终止值。

例如：

range(4,0,-1)产生的序列是 4、3、2、1。

range(8,-2,-3)产生的序列是 8、5、2、-1。

range(-1,-10,-2)产生的序列是-1、-3、-5、-7、-9。

使用 range()函数可以生成数字序列，而且可以使用单个参数、两个参数、三个参数。

【例 3-22】使用 range()函数产生等差数列——单个参数的应用：

```
for i in range(6):
   print(i)
```

执行以上代码，输出结果为：

```
0
1
2
3
4
5
>>>
```

【例 3-23】使用 range()函数产生指定区间的值——两个参数的应用：

```
for i in range(5,10) :
   print(i)
```

执行以上代码，输出结果为：

```
5
6
7
8
9
>>>
```

【例 3-24】使用 range()函数产生偶数序列——三个参数的应用：

```
for i in range(0, 10, 2) :
    print(i)
```

执行以上代码，输出结果为：

```
0
2
4
6
8
>>>
```

【例 3-25】range()函数的应用——参数为负数：

```
for i in range(-10, -100, -20) :
    print(i)
```

执行以上代码，输出结果为：

```
-10
-30
-50
-70
-90
>>>
```

也可以结合 range()和 len()函数以遍历一个序列的索引，如下例所示。

【例 3-26】遍历列表：

```
a = ['Lenovo','Google','Baidu','Tencent','Alibaba','Apple','SINA']
for i in range(len(a)):
    print(i, a[i])
```

执行以上代码，输出结果为：

```
0 Lenovo
1 Google
2 Baidu
3 Tencent
4 Alibaba
5 Apple
6 SINA
>>>
```

还可以使用 range()函数来创建一个列表，并且打印出来。

【例 3-27】创建列表：

```
print(list(range(5)))
```

执行以上代码，输出结果为：

```
[0, 1, 2, 3, 4]
>>>
```

在 Python 中不能声明矩阵，也不能列出维数，但可以利用列表中夹带列表的形式表示，即利用嵌套的列表来表示。

【例 3-28】生成 3×3 的矩阵：

```
count = 1
array = []
for i in range(0, 3):
    tmp = []
    for j in range(0, 3):
        tmp.append(count)
        count =count + 1
    array.append(tmp)
print (array)
```

执行以上代码，输出结果为：

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```

【例 3-29】矩阵初始化(全部元素赋 0 值)：

```
array = []
for i in range(0, 3):
    tmp = []
    for j in range(0, 3):
        tmp.append(0)
    array.append(tmp)
print (array)
```

执行以上代码，输出结果为：

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

尽管使用列表实现矩阵轻而易举，但借助第三方扩展库将更加方便、更加灵活，相关内容请读者参考第 9 章的例 9-4。

【例 3-30】编程实现排序功能：

```
a = [11, 21, 53, 32, 67, 82, 43]
for i in range(len(a) - 1, 0, -1):        #等价于 range(1, len(a), 1)
    for j in range(0, i):
        if a[j] > a[j + 1]:
            a[j], a[j + 1] = a[j + 1], a[j]
print (a)
```

执行以上代码，输出结果为：

```
[11, 21, 32, 43, 53, 67, 82]
>>>
```

说明如下。

① a 是一个乱序的列表。

② range(len(a)-1,0,-1)也就是 range(6,0,-1)，意思是从 6 到 0 间隔-1 产生序列，也就是倒序的 range(1,7,1)，随后把这些值依次赋给 i，那么 i 的值将会是[6, 5, 4, 3, 2,1]。

③ 第 2 个 for 是内层循环，j 的值将随 i 而变化，每次分别是[0, 1, 2, 3, 4, 5]、[0, 1, 2,

3, 4]、[0, 1, 2, 3]、[0, 1, 2]、[0, 1]、[0]。

④　比较第 j 个元素和第 j+1 个元素的大小。

⑤　若 a[j]大，则交换第 j 个元素和第 j+1 个元素(即前面的大时交换)。

⑥　打印排序后的列表。

本例程序所用的排序算法是一个很有名的算法，称作“冒泡排序”。如果不想公开本例所用的算法，则可以用 C 或 C++语言编写这一部分代码，再编写相应的封装接口，将其实现为 Python 中的一个模块，引入到 Python 程序中。因本书篇幅所限，具体方法请读者参考文献[27]。

事实上，我们完全没有必要通过编程来实现排序，因为 Python 内建了 sort()函数，可直接用于排序，这样一来，不仅使用方便，而且速度快，对数据类型的适应性也很强。

【例 3-31】直接调用 sort()函数实现排序：

```
a = [11, 21, 53, 32, 67, 82, 43]
a.sort()
print (a)
b = ['11', '21', '53', '32', '67', '82', '43']
b.sort()
print (b)
```

执行以上代码，输出结果为：

```
[11, 21, 32, 43, 53, 67, 82]
['11', '21', '32', '43', '53', '67', '82']
>>>
```

3.4　while 循环

Python 循环语句 while 的执行流程如图 3-3 所示。

3.4.1　while 循环基本结构

Python 中 while 循环的一般形式如下：

```
while <expression>:
    <statements_1>
```

同样需要注意冒号和缩进。另外，与 C 语言不同，在 Python 中没有 do…while 循环。

【例 3-32】使用 while 循环语句计算 1 到 n 的总和，这里 n=100。

```
n = int(input("输入一个数字:"))         #输入一个数，并转换成整数
sum = 0                              #求和变量初始化为 0
counter = 1                           #计数器初始化为 1
while counter <= n:                     #计数器<=100 时
    sum += counter                     #求和变量加上 counter
    counter += 1                       #计数器增 1
print("1 到%d 之和为: %d" % (n,sum))    #打印结果
```

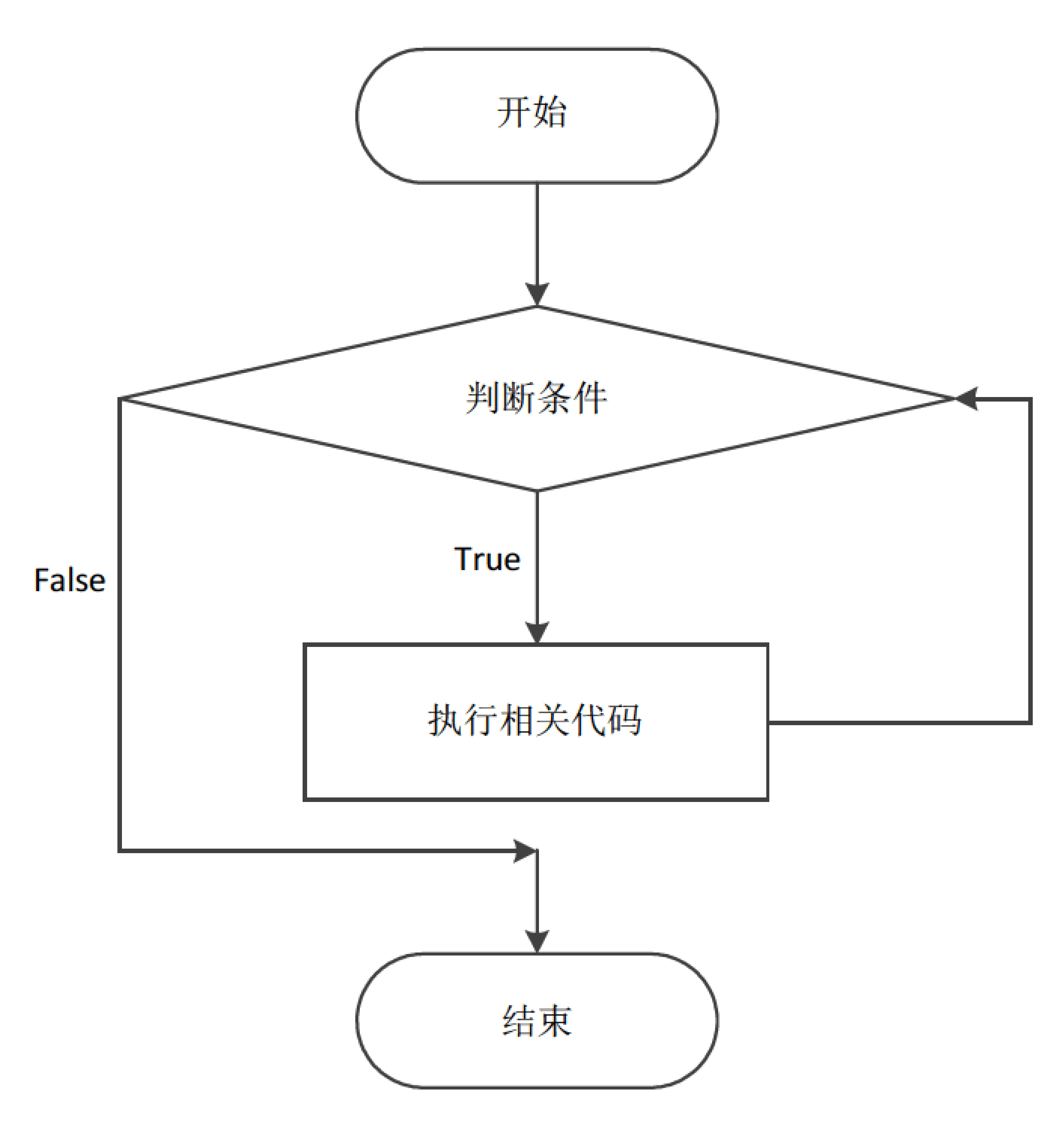


图 3-3　while 循环的执行流程

执行以上代码，输出结果为：

```
输入一个数字:100
1到100之和为: 5050
>>>
```

注意：使用复合运算符或称增量运算符(+=、-+、*+、/=、%=等)，代码显得更加紧凑、简洁，并且易读，读者应掌握这种用法。

【例 3-33】通过设置条件表达式永远为 True 来实现无限循环：

```
a = 1
while a == 1 :  #表达式永远为 True
    num = int(input("输入一个数字  :"))
    print ("你输入的数字是: ", num)
```

执行以上代码，输出结果如下：

```
输入一个数字  :1
你输入的数字是:  1
输入一个数字  :-1
你输入的数字是:  -1
输入一个数字  :1234567890987654321
你输入的数字是:  1234567890987654321
输入一个数字  :
Traceback (most recent call last):
  File "C:/Windows/System32/hh.py", line 3, in <module>
    num = int(input("输入一个数字  :"))
  File "C:\Users\Administrator\AppData\Local\Programs\Python\Python35-32\lib\idl
elib\PyShell.py", line 1389, in readline
    line = self._line_buffer or self.shell.readline()
KeyboardInterrupt
>>>
```

注意：

- 可以按 Ctrl+C 组合键来退出当前的无限循环，但退出时将引发 KeyboardInterrupt(键盘中断)异常。
- 无限循环对服务器上客户端的实时请求非常有用。

3.4.2　while 循环嵌套

Python 中 while 循环的嵌套形式如下：

```
while <expression1>:
   while <expression2>:
      <statements 1>
   <statements_2>
```

while 循环也可以在循环体内嵌入其他的循环体，如在 while 循环中可以嵌入 for 循环，反之，也可以在 for 循环中嵌入 while 循环。

【例 3-34】使用嵌套循环输出 20~50 之间的素数：

```
i = 20
while(i < 50):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j): print (i, " 是素数")
   i = i + 1
```

执行以上代码，输出结果如下：

```
23  是素数
29  是素数
31  是素数
37  是素数
41  是素数
43  是素数
47  是素数
>>>
```

3.4.3　while 循环中使用 else 分支

while … else 在条件语句为 False 时执行 else 后的语句块。

【例 3-35】else 分支在 while 循环中的使用：

```
cou = 0
while cou <= 3:
   print (cou, " 小于或等于 3")
   cou += 1
else:
   print (cou, " 大于 3")
```

执行以上代码，输出结果如下：

```
0   小于或等于 3
1   小于或等于 3
2   小于或等于 3
3   小于或等于 3
4   大于 3
>>>
```

类似 if 语句的语法，如果 while 循环体中只有一条语句，也可以将该语句与 while 写在同一行中，如下例所示。

【例 3-36】单行循环体：

```
var = 1
while (var): print ('hello Python')
print ("bye!")
```

本例程序中，循环体只有一行，与 while 写在了同一行上，可能书写起来比较方便，但这样的代码可读性差，希望读者将这行代码移到下一行并合理地缩进。另外一个原因是，如果需要添加新的代码，读者还是需要把它移到下一行，使用标准的形式。

执行以上代码，输出结果如图 3-4 所示。由于程序采用了死循环，不得不使用 Ctrl+C 中断执行，因而最后引发了与例 3-33 一样的 KeyboardInterrupt 异常。

```
hello Python
hello Python
hello Python
hello PythonTraceback (most recent call last):
  File "D:\Python34\text1.py", line 2, in <module>
    while (var): print ('hello Python')
  File "C:\Users\Administrator\AppData\Local\Progr
ams\Python\Python35-32\lib\idlelib\PyShell.py", li
ne 1347, in write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
>>>
```

图 3-4　例 3-36 的运行结果

3.4.4　break 和 continue 语句在循环中的使用

break 语句可以跳出 for 和 while 的循环体。如果程序从 for 或 while 循环中终止，则循环所对应的 else 块将不执行。

【例 3-37】break 语句在 for 循环中的使用：

```
for letter in 'helloworld':
   if letter == 'l':
      break
   print ('当前字母 :', letter)
```

执行以上代码，输出结果如下：

```
当前字母 : h
当前字母 : e
>>>
```

【例 3-38】break 语句在 while 循环中的使用：

```
var = 5
while var > 0:
    print ('当前变量值是 :', var)
    var -= 1
    if var == 3:
        break
```

执行以上代码，输出结果如下：

```
当前变量值是 : 5
当前变量值是 : 4
>>>
```

continue 语句告诉 Python 跳过当前循环体中的剩余语句，然后继续进行下一轮循环。

【例 3-39】continue 语句在 for 循环中的使用：

```
for letter in 'helloworld':
    if letter == 'l':
        continue
    print ('当前字母 :', letter)
```

执行以上代码，输出结果如下：

```
当前字母 : h
当前字母 : e
当前字母 : o
当前字母 : w
当前字母 : o
当前字母 : r
当前字母 : d
>>>
```

【例 3-40】continue 语句在 while 循环中的使用：

```
var = 5
while var > 0:
    var -= 1
    if var == 3:
        continue
    else:
        print ('当前变量值是 :', var)
```

执行以上代码，输出结果如下：

```
当前变量值是 : 4
当前变量值是 : 2
当前变量值是 : 1
当前变量值是 : 0
>>>
```

3.4.5　pass 在循环中的使用

Python 的 pass 是空语句，旨在保持程序结构的完整性。pass 不做任何事情，一般用作占位语句。

【例 3-41】pass 语句的使用：

```
for letter in 'hello':
```

```
    if letter == 'l':
        pass
        print ('执行 pass 语句')
    print ('当前字母:', letter)
```

执行以上代码，输出结果如下：

```
当前字母: h
当前字母: e
执行 pass 语句
当前字母: l
执行 pass 语句
当前字母: l
当前字母: o
>>>
```

3.4.6　end 在循环中的使用

默认情况下，print 语句输出时自动换行，使用 end=‘’，可将结果输出到同一行，或在输出的末尾添加其他字符。此功能前面已使用多次，这里再举一例，进一步说明其用法。

【例 3-42】打印 100 以内的斐波那契数列。

意大利数学家斐波那契曾研究一种递归数列，后人称其为斐波那契数列。该数列的前两项均为 1，从第三项开始，每一项都等于其前两项之和。代码如下：

```
a, b = 0, 1
while b <= 100:
    print(b, end=',')   #print(b)
    a, b = b, a+b
```

执行以上代码，输出结果如下：

```
1,1,2,3,5,8,13,21,34,55,89,
>>>
```

注意：

- 第一行包含一个复合赋值：变量 a 和 b 同时得到新值 0 和 1。最后一行再次使用同样的方法。
- 复合赋值语句右边的表达式会在赋值变动之前执行。右边表达式的执行顺序是自左向右的。
- 如果用注释中的打印语句，斐波那契数列会按一行一个数打印输出。

3.5　案例实训：输出所有和为某个正整数的连续正数序列

小明很喜欢数学，有一天，他在做数学作业时，题目要求计算出 9~16 的和，他马上就写出了正确答案：100。但是他并不满足于此，他在想，究竟有多少种连续的正数序列的和为 100(至少包括两个数)？没多久，他就得到另一组连续正数和为 100 的序列：18，19，20，21，22。现在我们使用 Python 编程，找出所有和为 S 的连续正数序列。

题目描述：输入一个正整数 S(S>2)，输出所有和为 S 的连续正整数序列。要求先输出符合要求的序列的数目，然后分行输出各个序列。

题目分析：1+2+…，“展右端”，即总是加最右端的数，直至其和等于输入数，或大于输入数；若等于输入数，则输出该序列；若大于输入数，则依次“砍左端”，即依次减掉最左端的数，直到序列和小于输入数。

算法如下。

(1) 输入数(大于 2 的正整数)。

(2) 找到输入数的 1/2，作为序列中值。

(3) 当前和=1+2。

(4) 当左端数>中值时，转(8)。

(5) 如果序列和=输入数，则输出该序列，然后“展右端”，求新序列的和，转(4)。

(6) 如果序列和>输入数，则“砍左端”，转(4)。

(7) 如果序列和<输入数，则“展右端”，求新序列的和，转(4)。

(8) 输出序列。

程序如下：

```
#求连续正数和问题
print("请输入大于 2 的正整数：")
tsum=int(input())
while(tsum<=2):
    print("请输入大于 2 的正整数：")
    tsum=int(input())
begin = 1  #首元素
end = 2    #尾元素
middle = (tsum + 1)>>1 #右移一位，获取中间值，相当于除以 2
curSum = begin + end  #连续序列的和(最小的连续序列和为 1+2=3，故从 3 开始)
output = [] #保存结果的列表(有几个序列，其中就有几个子列表)
while begin < middle:    #“展右端，砍左端”
    if curSum == tsum:  #若 curSum==tsum，则符合条件
        output.append([begin, end]) #将序列的首尾元素列表(2 元素)添加到 output
        end += 1   #为寻找下一个可能的序列作准备
        curSum += end
    elif curSum > tsum:  #若 curSum>tsum，则从 curSum 中减去 begin，begin 再增 1
        curSum -= begin
        begin += 1
    else:           #若 curSum<tsum，则 end 先增加 1，curSum 再加上 end
        end += 1
        curSum += end
#按照指定格式输出结果
if len(output)!=0:
    print("有{0}种序列：".format(len(output)))
    for i in range(0,len(output),1):
        print("序列{0}：".format(i+1))
        for j in range(output[i][0],output[i][1]+1):
            print(j,end='  ')
        print('\n')
else:
    print("没有满足条件的序列！！！")
```

执行两次以上代码，输出结果为：

```
请输入大于2的正整数：
100
有2种序列：
序列1：
9   10   11   12   13   14   15   16

序列2：
18   19   20   21   22

>>> ==========================
>>>
请输入大于2的正整数：
8
没有满足条件的序列！！！
>>>
```

程序中使用了一个嵌套的 output 列表，用于存储符合条件的序列，但存储的是该序列的首尾两个数，输出时则输出整个序列。

本 章 小 结

本章主要介绍与选择和循环相关的内容。首先介绍了 if、if-else、if-elif-else 三种语句的一般形式，介绍了三元运算符和比较操作符，以及 if 的嵌套用法；其次介绍了 for 循环的控制结构、一般格式及其嵌套使用；然后介绍了 range()函数的功能及各参数的用法，并列举了详尽的例子；接着介绍了 while 语句的一般形式、嵌套用法；最后介绍了如何在循环中使用 else 分支，以及 break、continue 和 pass 语句在循环中的使用等。

习　　题

1. 填空题

(1) 一般而言，if、while、for 语句的行末要使用(　　　)(标点符号)。

(2) 下列程序的输出结果是(　　　):

```
a=3
b=3
print(a is b)
```

(3) range(1,10,3)的值是(　　　)。

(4) [x**3 for x in [1,2,3,4,5]]的结果是(　　　)。

(5) 下列程序的运行结果是(　　　):

```
str1="E:\TJPU\YLH\TEST.TXT"
for i in str1:
    if i!='\\':
        print(i,end='')
```

2. 选择题

(1) 执行下列语句后的显示结果是(　　):

```
world="world"
print ("hello"+ world)
```

A. helloworld　　B. “hello” world

C. hello world　　D. 语法错误

(2) 下列 Python 语句中正确的是(　　)。

A. (min = x　if　x < y)　else　y　　B. max == x > y ? x : y

C. if (x > y)　　print x　　D. while True : pass

(3) 已知 x = 43，y = False；则表达式(x >= y and ‘A’ < ‘B’ and not y)的值是(　　)。

A. False　　B. 语法错　　C. True　　D. 假

(4) 以下程序的输出结果是(提示：ord(‘a’)==97)(　　):

```
lista = [1,2,3,4,5,'a','b','c','d','e']
print  (lista[2] + ord(lista[5]))
```

A. 100　　B. ‘d’　　C. d　　D. TypeError

(5) 下列哪个语句在 Python 中是非法的？(　　)

A. x = y = z = 1　　B. x = (y = z + 1)

C. x, y = y, x　　D. x　+=　y

(6) 下面的循环体执行的次数与其他不同的是(　　)。

A.
```
i = 0
while( i <= 100):
        print (i)
        i = i + 1
```

B.
```
for i in range(100):
        print (i)
```

C.
```
for i in range(100, 0, -1):
        print (i)
```

D.
```
i = 100
    while(i > 0):
        print (i)
        i = i - 1
```

3. 问答题

(1) 分析逻辑运算符 or 的短路求值特性。

(2) Python 中 pass 语句的作用是什么？

(3) 简述 Python 中 range()函数的用法。

(4) Python 中 break、continue 语句的作用是什么？

4. 实验操作题

(1) 编写输出 10 以内素数的循环程序。

(2) 使用 if-elif-else 语句判断输入的数字是正数、负数还是零。使用嵌套的 if 语句实现同样的功能。

(3) 用 if 语句判断输入的一个数字是奇数还是偶数。

(4) 用 if 语句判断用户输入的年份是否为闰年。

(5) 使用标准格式输出阶乘(factorial)。整数的阶乘是所有小于及等于该数的正整数的积，即：n!=1×2×3×... ×n。0 的阶乘定义为 1。

(6) 改进九九乘法表，用 for 语句和 range()函数实现。建议使用 end 换行。

(7) 求指定区间内的水仙花数(亦称阿姆斯特朗数)，要求使用循环语句和判断语句。

如果一个 n 位正整数等于其各位数字的立方之和，则称该数为水仙花数或阿姆斯特朗数。例如 3^3 + 7^3 + 0^3 = 370。1000 以内的水仙花数有：1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407。



第 4 章

函 数 模 块

本章要点

(1) Python 代码编写规范和风格。
(2) 函数的定义与调用。
(3) 函数参数的传递。
(4) Python 变量作用域。
(5) 函数与递归。
(6) 迭代器与生成器。
(7) Python 自定义模块。
(8) 输入输出语句的基本格式及执行规则。
(9) 匿名函数的定义与使用。

学习目标

(1) 了解“分治”的概念，熟悉 Python 的代码风格。
(2) 掌握构建自定义模块的方法，掌握模块化编程的程序设计方法。
(3) 理解递归函数的执行过程，掌握递归函数的编写方法。
(4) 全面深入地了解内建模块，并能运用编程的方法解决实际问题。

本章主要讨论程序的另一种结构：函数。

针对程序流程控制而言，函数的重要性与选择结构和循环结构是一样的。函数允许程序的控制在不同的代码片段之间切换。函数的重要意义在于可以在程序中清晰地分离不同的任务，将复杂的问题分解为若干个相对简单的子问题，并逐个解决，即“分而治之”，或称“分治”。函数允许我们按照这样的方式编写或阅读程序：首先关注所有任务，然后关注如何完成每项任务。换言之，函数允许我们按照“自顶向下、逐层细化”的理念编写程序。此外，函数还为代码重用提供了一种通用机制。

4.1　Python 代码编写规范

Python 最常用的编码风格还是 PEP8，先简单引用一下《Python 之禅》中的几句经典阐释。

(1) 优美胜于丑陋(Python 以编写优美的代码为目标)。
(2) 明了胜于晦涩(优美的代码应当是明了的，命名规范，风格相似)。
(3) 简洁胜于复杂(优美的代码应当是简洁的，不要有复杂的内部实现)。
(4) 复杂胜于凌乱(即使复杂是不可避免的，代码间也不能有难懂的关系，要保持接口简洁)。
(5) 扁平胜于嵌套(优美的代码应当是扁平的，不能有太多的嵌套)。
(6) 间隔胜于紧凑(优美的代码有适当的间隔，不要奢望一行代码解决所有问题)。
(7) 可读性很重要(优美的代码是可读的)。

4.1.1　Python 代码风格

1. 避免劣化代码

(1) 避免只用大小写来区分不同的对象。

(2) 避免使用容易引起混淆的名称，变量名应与所解决的问题域一致。

(3) 不要害怕变量名过长。

2. 代码中添加适当注释

(1) 行注释仅注释复杂的操作、算法，难理解的技巧，或不够一目了然的代码。

(2) 注释和代码要隔开一定的距离，无论是行注释还是块注释。

(3) 给外部可访问的函数和方法(无论是否简单)添加文档注释，注释要清楚地描述方法的功能，并对参数、返回值和可能发生的异常进行说明，使得外部调用的人仅看文档字符串(docstring)就能正确使用。

(4) 推荐在文件头中包含 copyright 声明、模块描述等。

(5) 注释应该是用来解释代码的功能、原因、想法的，不该对代码本身进行解释。

(6) 对不再需要的代码，应该将其删除，而不是将其注释掉。

3. 适当添加空行使代码布局更为优雅、合理

(1) 在一组代码表达完一个完整的思路之后，应该用空白行进行间隔，推荐在函数定义或者类定义之间空两行，在类定义与第一个方法之间，或需要进行语义分隔的地方空一行。空行是在不隔断代码之间内在联系的基础上插入的。

(2) 尽量保证上下文语义的易理解性，一般是调用者在上，被调用者在下。

(3) 避免过长的代码行，每行最好不要超过 80 个字符。

(4) 不要为了保持水平对齐而使用多余的空格。

4. 编写函数时的几个原则

(1) 函数设计要尽量短小，嵌套层次不宜过深。

(2) 函数声明应做到合理、简单、易于使用，函数名应能正确反映函数的大体功能，参数设计应简洁明了，参数个数不宜过多。

(3) 函数参数设计应考虑向下兼容。

(4) 一个函数只做一件事，每个函数都应有一个单一的、统一的目标，尽量保证函数语句粒度的一致性。

(5) 只有在真正必要的情况下才使用全局变量。

(6) 不要改变可变类型的参数，除非调用者希望这样做。

(7) 避免直接改变另一个文件模块中的变量。

5. 将常量集中到一个文件里

Python 没有提供定义常量的直接方式，一般有两种方法来使用常量。

(1) 通过命名风格来提醒使用者该变量代表的意义为常量，如常量名所有字母大写，

用下划线连接各个单词，如 MAX_NUMBER、PI 等。

(2) 通过自定义的类实现常量功能，常量要求符合两点，一是命名必须全部为大写字母，二是值一旦绑定便不可再修改。

4.1.2 例子说明

下面是一些小例子，用于简单说明 Python 的语法风格。

1. 交换两个变量

C 语言代码如下：

```
int a = 3, b = 5;
int tmp = a;
a = b;
b = tmp;
```

【例 4-1】Python 风格代码：

```
a, b = b, a
```

注意：*Python 语法风格追求的是对 Python 语法的充分发挥，写出的 Python 代码不是看着像 C 或 Java 代码。*

2. 不应过分使用技巧

【例 4-2】逆序输出字符串：

```
a = [1, 2, 3, 4, 5]
b = 'abcde'
print (a[::-1])
print (b[::-1])
```

执行以上代码，输出结果为：

```
[5, 4, 3, 2, 1]
['e', 'd', 'c', 'b', 'a']
>>>
```

该程序的作用就是输出逆序的 a 和 b，但还是比较晦涩难懂。Python 语法风格追求的是充分利用 Python 语法，故上面的代码可写为：

```
a = [1, 2, 3, 4, 5]
b = 'abcde'
print (list(reversed(a)))
print (list(reversed(b)))
```

3. 字符串格式化

本书 2.3 节介绍字符串时曾提及%s 的用法。一般来说，使用%s 进行字符串的格式化比较简洁，例如下面的例子。

【例 4-3】字符串的格式化输出：

```
name = 'Tim'
```

```
sex = 'male'
print ('Hello %s, your sex is %s !' % (name, sex))
```

执行以上代码，输出结果为：

```
Hello Tim, your sex is male !
>>>
```

其实，%s 是比较影响可读性的，尤其是存在多个%s 时，很难看清楚哪个占位符对应哪个实参。比较有 Python 风格的代码如下：

```
value = {'name': 'Tim', 'sex': 'male'}    #字典
print ('Hello %(name)s, your sex is %(sex)s !' % value)
```

使用%占位符的形式，依旧不是 Python 最推荐的，最具 Python 风格的代码如下：

```
print ('Hello {name}, your sex is {sex} !'.format(name = 'Tim', sex =
'male'))
```

str.format()是 Python 最为推荐的字符串格式化方法，4.3.3 小节将具体介绍。

4. 过多的 if...elif...elif...else 应使用字典来实现

【例 4-4】判断数字类型：

```
n = int(input("输入数字类型: "))
if n == 0:
   print ("类型 0")
elif n == 1:
   print ("类型 1")
elif n == 2:
   print ("类型 2")
else:
   print ("其他类型")
```

执行以上代码，输出结果为：

```
输入数字类型: 3
其他类型
>>>
```

上述代码使用字典来实现更好一些，更加简洁明了：

```
def f(x):
   return{
      0: "类型 0",
      1: "类型 1",
      2: "类型 2",
   }.get(x, "其他类型")
print(f(5))
```

4.2　自 建 模 块

Python 的自建模块一般体现为函数。Python 函数有如下特点：

- 函数是组织好的、可重复使用的，用来实现单一或相关联功能的代码段。
- 函数首先关注所有任务，然后关注如何完成每项任务。函数类型有两种：有返回

值的函数和仅仅执行代码而不返回值的函数。

- 函数能提高应用程序的模块化程度和代码的重用性。

Python 提供很多内建函数(亦称内置函数)，比如 print()、int()、float()等，但也可以自己创建函数，在 Python 中称为用户自定义函数。

4.2.1 定义一个函数

用户可以自己定义一个函数，用以完成某些功能。函数定义也常常称为函数声明。以下是函数定义应遵循的规则：

- 函数定义以 def 关键字开头，后接函数标识符名称和圆括号()，最后是冒号(:)。
- 函数命名应该能描述函数的功能，而且必须符合标识符的命名规则。
- 函数的形式参数(形参)必须放在圆括号中。
- 函数的第一行语句可以选择性地使用文档字符串，用于存放函数说明。
- 函数内容(语句块)放于冒号后，每条语句都要缩进相应数量的空格。
- 以 return [表达式]结束函数，选择性地返回一个值给调用者。不带表达式的 return 相当于返回 None。

定义 Python 函数的一般格式如下：

```
def functionName (parameter 1, parameter 2, parameter 3):
    functionBody
```

def 是 define 的缩写，functionName 是由用户命名的函数名，后面的参数表中可以有零个或多个形参。如果有形参，则函数调用时，函数名后面的括号中一般要提供实际参数(实参)。默认情况下，实参和形参是按函数声明中定义的顺序匹配的。当实参是一个表达式时，先要计算表达式的值，然后将该值传递给形参。相邻两个函数定义之间尽量用空行分隔。

【例 4-5】无参函数：

```
def hello() :
    "This is a function without parameter."
    print("Hello Python!")
hello()
```

执行以上代码，输出结果为：

```
Hello Python!
>>> |
```

【例 4-6】带参函数(一个或多个参数)：

```
def print wel(name):
    "This is a function with a parameter."
    print("Welcome", name)
print wel("Python")
def rectangle area(wide, high):
    "This is a function with two parameters."
    return wide * high
w = 2
```

```
h = 3
print("wide =", w, "high =", h, " area =", rectangle_area(w, h))
```

执行以上代码，输出结果为：

```
Welcome Python
wide = 2 high = 3  area = 6
>>>
```

注意：

- 在被调用前，函数定义必须先由 Python 解释器进行处理。
- 文档字符串可有可无。当文档字符串跨多行时，一般使用三引号作定界符。

4.2.2　函数调用

定义一个函数，给函数一个名称，并且指定函数里包含的形参和代码块结构，这个函数的基本结构就已经完成，这时可以通过另一个函数调用执行，也可以直接从 Python 命令提示符执行。

【例 4-7】定义函数：

```
def printme( str ):  #打印任何传入的字符串
   print (str)
   return
#调用函数
printme("调用自定义函数!")
printme("再次调用自定义函数!")
```

执行以上代码，输出结果为：

```
调用自定义函数!
再次调用自定义函数!
>>>
```

直接从 Python 命令提示符也可以调用自定义函数，执行过程如下：

```
>>> printme("从命令行直接调用自定义函数!")
从命令行直接调用自定义函数!
>>> |
```

注意：函数是可以嵌套定义的，但通常情况下不这样使用。

【例 4-8】函数嵌套定义：

```
def fun1():
   x = 5
   def fun2():
      print (x)
   fun2()
#调用自定义函数
fun1()
```

执行以上代码，输出结果为：

```
5
>>>
```

4.2.3 按引用传递参数

在 Python 中，所有形参(变量)都是按引用传递的。如果在函数里修改形参，那么在调用这个函数的函数里，实参(如果是变量的话)也被改变了。

【例 4-9】函数参数的传递及其“副作用”：

```
def printme( mylist ):
   mylist.append([1,2,3])
   print ("函数内的值: ", mylist)
   return
#调用 printme 函数
mylist = [5,15,25]
printme( mylist )
print ("函数外的值: ", mylist)
```

执行以上代码，输出结果为：

```
函数内的值:  [5, 15, 25, [1, 2, 3]]
函数外的值:  [5, 15, 25, [1, 2, 3]]
>>>
```

第 2 章中曾提到，参数的传递都是对象的引用，因此，实参 mylist 与形参 mylist 引用的是同一个对象，它们的地址是一样的。正因为如此，printme()函数将形参 mylist 的值改变了，实参 mylist 的值自然也就改变了(即产生了“副作用”)。这一点与 C 语言截然不同，使用时应特别注意。

注意： 本例程序中，形参和实参恰巧用了相同的标识符。其实，即使使用不同的标识符，得到的结果也是一样的，读者不妨一试。

4.2.4 参数类型

以下是调用函数时可使用的参数类型：

- 必需参数。
- 关键字参数。
- 默认参数。
- 不定长参数。

1. 必需参数

必需参数须以正确的顺序传入函数。调用时的数量必须与声明时的一样。例如调用printme()函数时，必须传入一个参数，不然会出现语法错误。

【例 4-10】使用必需参数：

```
def printme( str ):
   print (str)
   return
#调用 printme 函数
printme("apple")
```

执行以上代码，输出结果为：

```
apple
>>>
```

2. 关键字参数

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。下例程序在调用函数 printme()时使用了关键字参数。

【例 4-11】使用关键字参数：

```
def printme( str ):
    print (str)
    return
#调用 printme 函数
printme( str = "hello world" )
```

执行以上代码，输出结果为：

```
hello world
>>>
```

【例 4-12】使用关键字参数的实参不需要指定顺序：

```
#自定义函数
def printme( name, sex ):
    print ("名字: ", name)
    print ("性别: ", sex)
    return
#调用 printme 函数
printme( sex="female", name="Mary" )
```

执行以上代码，输出结果为：

```
名字:  Mary
性别:  female
>>>
```

3. 默认参数

调用函数时，如果没有传递参数，则 Python 会使用默认参数值，如下例中，如果没有传入 age 参数，则使用默认值 35。

【例 4-13】使用默认参数：

```
#自定义函数
def printme( name, age = 35 ):
    print ("名字: ", name)
    print ("年龄: ", age)
    return
#调用 printime 函数
printme( age=50, name="Mary" )
printme( name="Mary" )
```

执行以上代码，输出结果为：

```
名字： Mary
年龄： 50
名字： Mary
年龄： 35
>>>
```

4. 不定长参数

有时可能要求一个函数能处理比当初声明时更多的参数，这些参数叫作不定长参数。与上述两种参数不同，函数声明时不会在函数名后面的括号内指定参数的个数。

基本语法如下：

```
def functionname([formal args,] *var args tuple ):
   function suite
   return [expression]
```

加星号(*)的变量名会存放所有未命名的变量参数。如果在函数调用时没有指定参数，它就是一个空元组，这意味着调用这类函数时可以不向未命名的变量传递参数。

【例 4-14】使用不定长参数：

```
#自定义函数
def printme( arg1, *variable ):
   print (arg1)
   for var in variable:
      print (var)
   return
#调用 printme 函数
printme( 1 )
printme( 4, 3, 2 )
```

执行以上代码，输出结果为：

```
1
4
3
2
>>>
```

4.2.5 return 语句

return [表达式]语句用于退出函数，并选择性地向调用者返回一个表达式。如前所述，不带参数值的 return 语句返回的是 None。先前的例子都没有示范如何返回数值，下例对此做了示范。

【例 4-15】使用 return 语句返回数值：

```
#自定义函数
def sum( arg1, arg2 ):
   total = arg1 + arg2
   return total
#调用 sum 函数
total = sum( 110, 20 )
print (total)
```

执行以上代码，输出结果为：

```
130
>>>
```

4.2.6　变量的作用域

Python 中的变量并不是在哪个位置都可以访问的，访问权限取决于这个变量是在哪里赋值的。变量的作用域决定在哪一部分程序中可以访问哪个特定的变量。根据变量的作用域可把变量分为两种基本类型：

- 全局变量。
- 局部变量。

1. 全局变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外部的变量拥有全局作用域。局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名都将被加入到作用域中。

【例 4-16】全局变量与局部变量：

```
total = 0                          #total 在这里是全局变量
#自定义函数
def sum( arg1, arg2 ):
    total = arg1 + arg2                #total 在这里是局部变量
    print ("函数内是局部变量 : ", total)
    return total
#调用 sum 函数
sum( 2, 5 )
print ("函数外是全局变量 : ", total)
```

执行以上代码，输出结果为：

```
函数内是局部变量 :  7
函数外是全局变量 :  0
>>>
```

2. 变量的作用域和命名空间

变量是拥有匹配对象的名字(标识符)。命名空间是一个包含变量名称(键)和它们各自相应的对象(值)的字典。一个 Python 表达式可以访问局部命名空间和全局命名空间里的变量。如果一个局部变量和一个全局变量重名，则局部变量会覆盖全局变量。

每个函数都有自己的命名空间，类的方法的作用域规则与一般函数的一样(第 6 章有关于类的介绍)。

Python 会智能地猜测一个变量是局部的还是全局的，它假设任何在函数内赋值的变量都是局部的。因此，如果要在一个函数里给全局变量赋值，就必须使用 global 语句。

global VarName 的表达式会告诉 Python，VarName 是一个全局变量，这样 Python 就不会在局部命名空间里寻找这个变量。

例如，如果在全局命名空间里定义一个变量 money，再在函数内给变量 money 赋值，

然后 Python 会假定 money 是一个局部变量。然而并没有在访问前声明一个局部变量 money，结果就会出现一个 UnboundLocalError 错误。

【例 4-17】全局变量标识：

```
a = 20  #全局变量
def Add():
    global a  #全局变量标识
    a = a + 1
print (a)
Add()
print (a)
```

执行以上代码，输出结果为：

```
20
21
>>>
```

读者若想进一步理解作用域与命名空间的概念，可参见本书 6.1.3 节。

4.2.7 函数与递归

递归是一种直接或间接地调用自身的过程。在编程时，递归对解决一大类问题是十分有效的。

递归的特性有三点。

(1) 必须有一个明确的结束条件。

(2) 每次进入更深一层的递归时，问题规模相比上次递归应有所减少。

(3) 递归效率不高，递归层次过多会导致栈溢出(在计算机中，函数调用是通过栈这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧；每当函数返回，栈就会减一层栈帧。因为栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出)。

下面我们展示一下如何用递归来解决实际问题。

1. 阶乘

正整数的阶乘 n!是所有不大于该数的正整数之积。0 的阶乘定义为 0。

例如 5!= 5×4×3×2×1，得到的积是 120，即 5!=120。对于正整数 n，n!= n×(n-1)×(n-2)×…×1，也可以表示为递归的形式：

$$
\begin{aligned}
n! &= n\times(n-1)\times(n-2)\times(n-3)\times(n-4)\times\ldots\times2\times1 \\
&= n\times(n-1)!
\end{aligned}
$$

【例 4-18】阶乘的递归实现：

```
def Fac(num):
    if num <= 1: return 1
    return num * Fac(num-1)
print(Fac(5))
```

执行以上代码，输出结果为：

```
120
>>>
```

2. 斐波那契数列

第 3 章 3.4.6 节曾介绍了斐波那契数列。实际上，该数列是一种递归数列。当 n>1 时，该数列第 n 项等于其前面两项之和，所以，斐波那契数列可以写成如下的递归形式：

$$F(n)\begin{cases}1 & n=0.1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

【例 4-19】斐波那契数列递归算法：

```
def item( num ):
    if num == 0 :
        fi = 1
    elif num == 1:
        fi = 1
    else:
        fi = item ( num - 1) + item (num -2)
    return fi

def Fib( n ):
    i = 0
    while i < n:
        print (item(i),end=',')
        i += 1
Fib( 8 )
```

执行以上代码，输出结果为：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
>>>
```

以上递归方法的递归次数太多时，效率低下，所以通常可以采用递推算法，通过列表来实现。

【例 4-20】斐波那契数列递推算法：

```
def Fib(num):
    """递推算法"""
    fib = [1]          #用列表保存中间结果，初始值为1
    for i in range(num):
        if i < 2:fib.append(i + 1)
        else:fib.append(sum(fib[-2:]))
    return fib
print(Fib(10))
```

执行以上代码，输出结果为：

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> |
```

3. 整数因子分解

大于 1 的正整数 n 都可以分解为 $n = x_1*x_2*\ldots*x_m$。例如：当 n=12 时，共有 8 种不同的分解式：

```
12 = 12
```

```
12 = 6×2
12 = 4×3
12 = 3×4
12 = 3×2×2
12 = 2×6
12 = 2×3×2
12 = 2×2×3
```

对于给定的正整数 n，要求计算 n 共有多少种不同的分解式，这就是所谓的“整数因子分解”。

【例 4-21】正整数 35 的因子分解：

```
count=0
def fac(num):
    global count
    n = num +1
    A=[]
    for i in range(2,n):
        if num % i==0:
            A.append(i)
            l=len(A)
    for i in range(0,l):
        if int(num/A[i])==1:
            count += 1
        else:
            fac(int(num/A[i]))
    return count
if  name   == "  main  ":
print (fac(35))
```

执行以上代码，输出结果为：

```
3
>>>
```

因为整数 35 有三种因子分解方法，即 35=35、35=5×7、35=7×5，所以输出结果为 3。

注意：关于语句 if __name__ == “__main__”: 的几点说明：

- 一般来说，用 Python 写的文件既可以自身运行，又可以作为模块，被其他的程序调用。
- 当程序自身运行时，其__name__的值就是字符串“__main__”；如果是被其他程序调用，那么其__name__的值就不再是字符串“__main__”。这个判断的作用就是使其后的代码块只有在自身运行的情况下才执行，如果只是被调用，那么就不会执行了。
- 简单地说，这条语句的主要功能在于保留了一个脚本独立运行的能力，又同时使该脚本的功能函数与类能够成为其他脚本的拓展(在 6.3.3 小节中，还要对此做详细的介绍)。
- 这条语句通常可以用来给一个模块做测试，在项目整体运行的时候测试的代码将不会被执行。

4. 快速排序

快速排序(quicksort)是对冒泡排序的一种改进，由 C. A. R. Hoare 在 1962 年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序。

【例 4-22】快速排序：

```
def quicksort(array):
    """快速排序算法"""
    less = []
    more = []
    if len(array) <= 1:
        return array
    p = array.pop()
    for x in array:
        if x > p:
            more.append(x)
        else:
            less.append(x)
    return quicksort(less) + [p] + quicksort(more)
array1 = [21,32,42,53,34,25,57,54,78,89]
print (quicksort(array1))
```

执行以上代码，输出结果为：

```
[21, 25, 32, 34, 42, 53, 54, 57, 78, 89]
>>> |
```

5. 归并排序

归并排序使用二分法，其归根到底的思想是分治。用列表存储数据，将其不停地分为左边和右边两份，然后依此递归地分下去，再将它们按照两个有序列表合并起来。

【例 4-23】归并排序：

```
def mergeSort(list1):
    if len(list1) <= 1:
        return list1
    mid = int (len(list1)/2)
    left = mergeSort(list1[0:mid])
    right = mergeSort(list1[mid:len(list1)])
    return merge(left, right)

def merge(l, r):
    c = []
    i,j = 0,0
    while j < len(l) and i < len(r):
        if l[j] >= r[i]:
            c.append(r[i])
            i = i + 1
        else:
            c.append(l[j])
```

```
            j = j + 1
    for i in (l[j:] if i == len(r) else r[i:]):
        c.append(i)
    return c
if __name__ == '__main__':
    list2 = [-4, 0, 82, 3, 56, 19,-12,1]
print (mergeSort(list2))
```

执行以上代码，输出结果为：

```
[-12, -4, 0, 1, 3, 19, 56, 82]
>>> |
```

4.2.8 迭代器与生成器

1. 迭代器

迭代器(iterator)是 Python 最强大的功能之一，是访问集合元素的一种方式。迭代器是一个可以记住遍历位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完为止。迭代器只能向前，不会后退。迭代器有两个基本的方法：iter()和next()。字符串、列表或元组对象都可用于创建迭代器。

【例 4-24】迭代器示例：

```
list=["12","23","34","45"]
it = iter(list)       #创建迭代器对象
print (next(it))     #输出迭代器的下一个元素
print (next(it))     #输出迭代器的下一个元素
print (next(it))     #输出迭代器的下一个元素
print (next(it))     #输出迭代器的下一个元素
string = "hello"
st = iter(string)     #创建迭代器对象
print (next(st))     #输出迭代器的下一个元素
print (next(st))     #输出迭代器的下一个元素
tup1 = ('Google', 'Run', 1997, 2017)
tu = iter(tup1)      #创建迭代器对象
print (next(tu))     #输出迭代器的下一个元素
print (next(tu))     #输出迭代器的下一个元素
```

执行以上代码，输出结果为：

```
12
23
34
45
h
e
Google
Run
>>>
```

注意：迭代器对象可以使用常规的 for 语句进行遍历。

【例 4-25】用 for 循环遍历迭代器：

```
list=["12","23","34","45"]
it = iter(list)      #创建迭代器对象
for x in it:
    print (x, end="   ")

string = "hello"
st = iter(string)     #创建迭代器对象
for x in st:
    print (x, end="   ")

tupl = ('Google', 'Run', 2017, 2018)
tu = iter(tupl)     #创建迭代器对象
for x in tu:
    print (x, end="   ")
```

执行以上代码，输出结果为：

```
12   23   34   45   h   e   l   l   o   Google   Run   2017   2018
>>> |
```

注意：也可以在循环中使用 next()函数。

【例 4-26】用 while 循环遍历迭代器：

```
string = "hello"
st = iter(string)     #创建迭代器对象
i = 0           #循环控制变量
while (i < 5):
    print (next(st),end="   ")
    i = i + 1
```

执行以上代码，输出结果为：

```
h   e   l   l   o
>>>
```

2. 生成器

在 Python 中，使用 yield 的函数被称为生成器(generator)。跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作。可以更简单地理解：生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 yield 时，函数会暂停并保存当前所有的运行信息，返回 yield 的值，并在下一次执行 next()方法时从当前位置继续运行。

【例 4-27】生成器示例：

```
def fib(n):        #生成器函数 - 斐波那契
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
```

```
f = fib(15)        #f 是一个迭代器，由生成器返回生成
i = 0
while i < 15:
    print (next(f), end=" ")
i = i + 1
```

执行以上代码，输出结果为：

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
>>> |
```

yield 的作用就是把一个函数变成一个生成器函数，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个生成器函数。

生成器函数和普通函数的执行流程不一样。函数是顺序执行，遇到 return 语句或者执行完最后一条语句就返回。而变成生成器的函数，在每次调用 next()的时候执行，遇到 yield 语句返回，再次执行时从上次返回的 yield 语句处继续执行。

如果没有“yield a”语句，fib()函数就不是一个生成器函数，就无法返回迭代器，因而 while 循环中的 next()函数就无法使用，从而打印错误信息“TypeError: ‘NoneType’ object is not an iterator”。

4.2.9 自定义模块

Python 脚本可以在其集成开发环境 IDE 下运行，但如果从 IDE 中退出再进入，那么定义的所有方法和变量都会消失。为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器使用，这个文件被称为模块。这一概念在第 2 章的 2.1 节最后已经提及。

模块具有如下特点：

- 模块让使用者能够有逻辑地组织自己的 Python 代码段。
- 把相关的代码分配到一个模块里能让代码更好用、更易懂。
- 模块也是 Python 对象，具有随机的名字属性用来绑定或引用。
- 模块是一个包含所有已经定义的函数和变量的文件，模块里面也能够包含可执行代码。
- 模块的后缀名是.py，模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 Python 标准库的方法。

一个叫 function 的模块里的 Python 代码一般都能在一个叫 function.py 的文件中找到。

【例 4-28】一个简单的模块 function.py：

```
def Add( arg1, arg2 ):
    total = arg1 + arg2
    return total
def Sub( arg1, arg2 ):
    diff = arg1 - arg2
    return diff
def printme( str ):
    print (str)
    return
```

1. import 语句

如果需要使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[,... moduleN]]
```

当解释器遇到 import 语句时，模块当前搜索路径就会被导入到 Python 解释器。搜索路径是一个解释器先进行搜索的所有目录的列表。如果想要导入模块 function.py，需要把 import 命令放在脚本的顶端。例如：

```
import function                  #导入模块 function.py
function.printme("hello Python") #调用模块里包含的函数，不能漏写“function.”
```

执行以上代码，输出结果为：

```
hello Python
>>> |
```

2. from…import 语句

Python 的 from…import 语句让使用者从模块中导入一个指定的部分到当前命名空间中，其语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，如果需要导入模块 function 中的 Add 函数，则可使用语句：

```
from function import Add
```

这个声明不会把整个 function 模块导入到当前的命名空间中，它只会将 function 里单个的 Add 函数导入到执行这个声明的模块的全局符号表中。如：

```
from function import printme, Sub, Add  #仅导入 printme、Sub、Add 这三个函数
from function import *    #导入 function 中的全部对象(包括函数、类等)
```

应尽量少用 from module import *，因为判定一个特殊的函数或属性是从哪来的有些困难，并且会使调试和重构更困难。

3. 定位模块

当导入一个模块时，Python 解析器对模块位置的搜索顺序是：

- 当前目录。
- 如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。
- 如果找不到，Python 会察看默认路径。Linux 下的默认路径一般为/usr/local/lib/python/。Windows 下的模块搜索路径存储在 system 模块的 sys.path 变量中，如图 4-1 所示。变量里包含当前目录、PYTHONPATH，以及由安装过程决定的默认目录。

4. PYTHONPATH 变量

作为环境变量，PYTHONPATH 由装在一个列表里的许多目录组成。PYTHONPATH 的语法和 shell 变量 PATH 是一样的。

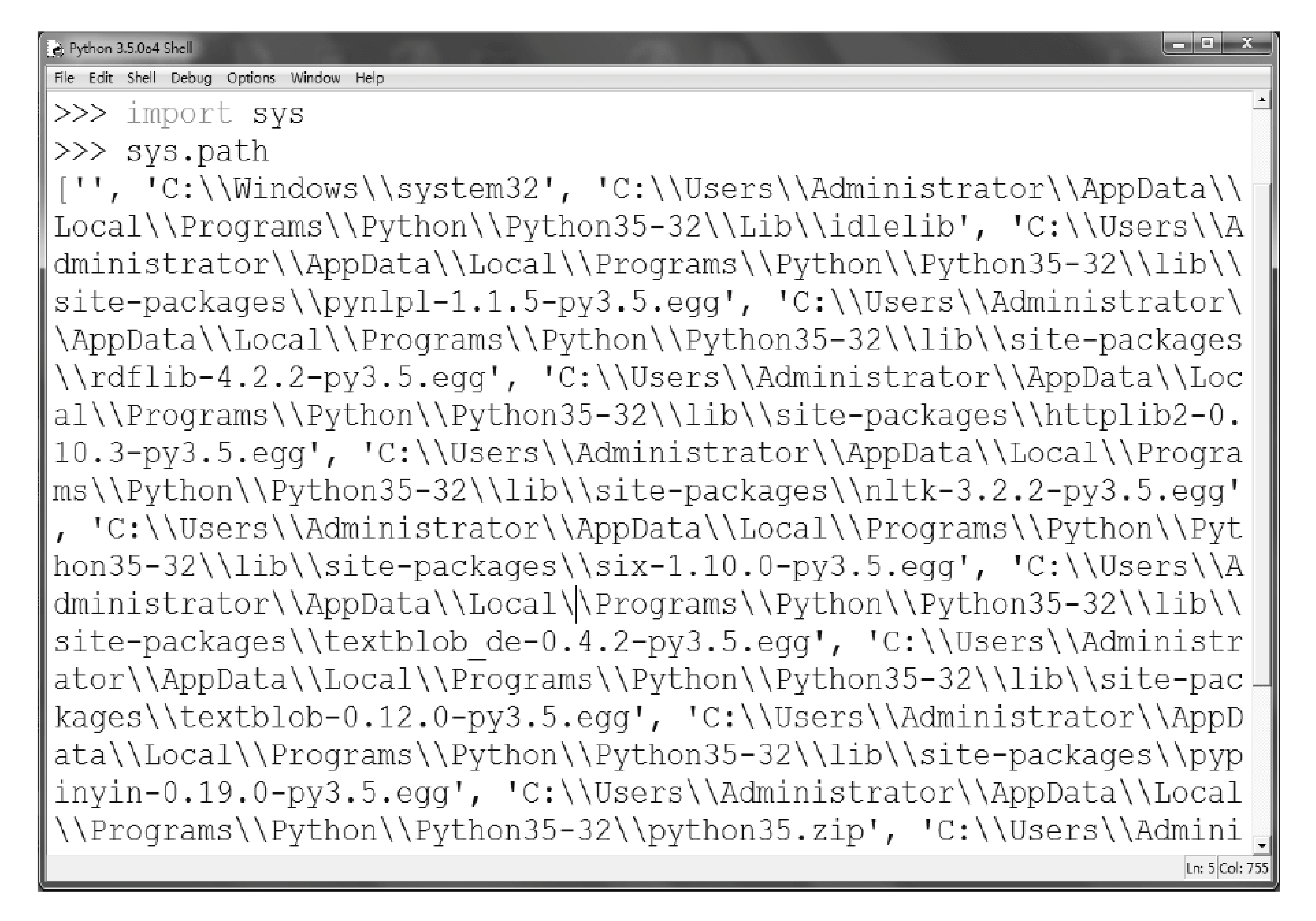


图 4-1 Windows 中的模块路径示例

在 Windows 系统中，典型的 PYTHONPATH 如下：

```
set PYTHONPATH=c:\python34\lib
```

在 Linux 系统中，典型的 PYTHONPATH 如下：

```
set PYTHONPATH=/usr/local/lib/python
```

4.3 标 准 模 块

Python 本身自带一些标准的模块库，有些模块直接被构建在解析器里，这些虽然不是 Python 语言内建的功能，但是它却能很高效地使用，甚至是系统级调用也没问题。这些组件会根据不同的操作系统进行不同形式的配置，比如，winreg 这个模块就只会提供给 Windows 系统。

4.3.1 内建函数

range()函数为内建函数的一种，前面已经讲过，此处不再赘述。

部分内建函数如表 4-1 所示。

表 4-1 Python 的部分内建函数

内建函数	实　例
int()	int(2.34) 结果是 2
chr()	chr(67) 结果是 C

续表

内建函数	实　例
ord()	ord('B') 结果是 66
round()	round(67.12345,2) 结果是 67.12

4.3.2　读取键盘输入

Python 用于读取键盘输入的内建函数是 input()，前面已多次用到，它是基本的 I/O 函数，用于从标准输入设备读入一行文本(默认的标准输入设备是键盘)。input()函数可以接收一个 Python 表达式作为输入，并将运算结果返回。

【例 4-29】基本输入输出：

```
str = input("请输入内容：")
print ("你输入的内容是：", str)
```

执行以上代码，输出结果为：

```
请输入内容：早安，天津
你输入的内容是：　早安，天津
>>>
```

注意： input()函数只能返回字符串，如需转换为其他类型，可以使用相应的转换函数，如 int()、float()等。

4.3.3　输出到屏幕

Python 有两种输出值的方式：print()函数和表达式语句。

最简单的输出方法是用 print()函数，可以给它传递零个或多个用逗号隔开的表达式。此函数把传递的表达式转换成一个字符串表达式，并将结果写到标准输出设备(屏幕)上。

【例 4-30】基本输出：

```
print ("hello,Python")
```

执行以上代码，输出结果为：

```
hello,Python
>>>
```

如果使用者希望将输出的值转换成字符串，可以使用 format()或 repr()函数。

- format()函数：返回一个用户易读的表达形式。
- repr()函数：产生一个解释器易读的表达形式。

1. format()函数的使用

如果希望输出的形式更加多样，可以使用 str.format()函数来格式化输出内容。现通过几个例子来说明 str.format()函数的用法。

【例 4-31】简单的格式化输出：

```
print('{}: "{}!"'.format('hello', 'Python'))
```

执行以上代码，输出结果为：

```
hello： "Python!"
>>>
```

注意： 花括号及其里面的字符(称作格式化字段)将会被 format()中的参数替换。在花括号中可以放入数字，用于指向传入对象在 format()中的位置，如下例所示。

【例 4-32】带有对象传入顺序的格式化输出：

```
print('{0} 和 {1}'.format('Google', 'Apple'))
print('{1} 和 {0}'.format('Google', 'Apple'))
```

执行以上代码，输出结果为：

```
Google 和 Apple
Apple 和 Google
>>>
```

如果在 format()中使用关键字参数，那么它们的值会指向使用该名字的参数。

【例 4-33】使用关键字参数的格式化输出：

```
print('{name}网址： {site}'.format(name='百度', site='www.baidu.com'))
```

执行以上代码，输出结果为：

```
百度网址： www.baidu.com
>>>
```

注意： 位置及关键字参数可以任意结合。

【例 4-34】同时使用对象传入顺序和关键字参数的格式化输出：

```
print('网站 {0}, {1}, 和 {other}。'.format('baidu',
'meituan',other='taobao'))
```

执行以上代码，输出结果为：

```
网站 baidu, meituan, 和 taobao。
>>>
```

注意： 可选项':'和格式标识符可以跟着字段名，这就允许对值更好地进行格式化。

下面的例子将圆周率 PI 保留到小数点后三位。

【例 4-35】使用格式标识符的格式化输出：

```
import math
print(' PI 的值近似是 {0:.2f}。'.format(math.pi))
```

执行以上代码，输出结果为：

```
 PI 的值近似是 3.14。
>>>
```

在 ':' 后传入一个整数，可以保证该域至少有这么多的宽度，在美化表格时很有用。

【例 4-36】表格式格式化输出：

```
tables = {'baidu': 1, 'meituan': 2, 'taobao': 3}
for name, number in tables.items():
   print('{0:8} -----> {1:8d}'.format(name, number))
```

执行以上代码，输出结果为：

```
taobao   ----->     3
meituan  ----->     2
baidu    ----->     1
>>>
```

本书 2.3 节已提到，使用%操作符也可以实现字符串的格式化：%前面的字符串与 C 语言中 printf()函数的格式化字符串完全相同；%后面的各项(超过一项时用元组)与前面的格式化操作符一一匹配后，返回格式化后的字符串。现再举一例。

【例 4-37】使用%操作符的格式化输出：

```
import math
print('常量 PI 的值近似是：%4.2f。' % math.pi)
```

执行以上代码，输出结果为：

```
常量 PI 的值近似是：3.14。
>>>
```

注意： "4"表示输出一共有 4 位，"2"表示小数点后有 2 位数字。但这种是旧格式，Python 推荐使用 str.format()。

2. repr()函数的使用

repr()将字符串转化为供解释器读取的形式。repr()的输出对 Python 比较友好，返回的是一个对象的"官方"字符串表示。

【例 4-38】repr()函数示例：

```
print (repr('Hello,Python.'))
print (repr(0.1))
x = 13 * 3.25
y = 20 * 34
s = 'x is ' + repr(x) + '||||y is ' + repr(y)
print (s)
print (repr('hello,Python\n'))
print (repr((x, y, ('word', 'world'))))
print (repr('Hello'))
obj='Hello,Python.'
print (obj==eval(repr(obj)))
```

执行以上代码，输出结果为：

```
'Hello,Python.'
0.1
x is 42.25||||y is 680
'hello,Python\n'
(42.25, 680, ('word', 'world'))
'Hello'
True
>>>
```

4.3.4　内建模块

Python 标准库中提供了不少内建模块，这些内建模块中又包含了很多实用的内建函

数。下面是使用 Python 标准库中模块的几个例子。

1. 时间(time)模块

Python 程序能用很多方式处理日期和时间，其中转换日期格式是一个常见的功能。Python 提供的 time 和 calendar 模块可以用于格式化日期和时间。时间间隔是以秒为单位的浮点小数，每个时间戳都是用从 1970 年 1 月 1 日 0 时 0 分 0 秒至今经过多长时间来表示的。Python 的 time 模块下有很多函数可以转换常见的日期格式，如函数 time.time()用于获取当前时间戳。

【例 4-39】 time 模块的使用：

```
import time  #引入 time 模块
ticks = time.time()
print ("当前时间戳为:", ticks)
```

执行以上代码，输出结果为：

```
当前时间戳为: 1501155583.150937
>>>
```

time 模块包含不少内建函数，既有与时间处理相关的函数，也有转换时间格式的函数，如表 4-2 所示。

表 4-2　time 模块的部分内建函数

序号	函数及描述
1	time.altzone 返回格林尼治西部的夏令时地区的偏移秒数。如果该地区在格林尼治东部则会返回负值(如西欧，包括英国)。对夏令时启用地区才能使用
2	time.asctime([tupletime]) 接受时间元组并返回一个可读的形式为例如“Tue Dec 11 18:07:14 2008”(2008 年 12 月 11 日周二 18 时 07 分 14 秒)的 24 个字符的字符串
3	time.clock() 用以浮点数计算的秒数返回当前的 CPU 时间。用来衡量不同程序的耗时，这个函数比 time.time()更有用
4	time.ctime([secs]) 作用相当于 asctime(localtime(secs))，未给参数相当于 asctime()
5	time.gmtime([secs]) 接收时间戳(从 1970 起算经过的浮点秒数)并返回格林尼治天文时间下的时间元组(假设为 t)。注：t.tm_isdst 始终为 0
6	time.localtime([secs] 接收时间戳(从 1970 起算经过的浮点秒数)并返回当地时间下的时间元组 t(t.tm_isdst 可取 0 或 1，取决于当地当时是不是夏令时)
7	time.mktime(tupletime) 接受时间元组并返回时间戳(从 1970 起算经过的浮点秒数)

续表

序号	函数及描述
8	time.sleep(secs) 推迟调用线程的运行，secs 指秒数
9	time.strftime(fmt[,tupletime]) 接收一个时间元组，并返回以可读字符串表示的当地时间，格式由 fmt 决定
10	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y') 根据 fmt 的格式把一个时间字符串解析为时间元组
11	time.time( ) 返回当前时间的时间戳(从 1970 起算经过的浮点秒数)
12	time.tzset() 根据环境变量 TZ 重新初始化时间相关设置

2. 日历(calendar)模块

此模块中的函数都是与日历相关的，例如打印某月的字符月历。这里我们约定，星期一是默认的每周第一天，星期天是默认的最后一天。calendar 模块有很多方法用来处理年历和月历，例如打印某月的月历。

【例 4-40】获取某月日历：

```
import calendar
cal = calendar.month(2017, 3)
print ("以下输出 2017 年 3 月份的日历:")
print (cal)
```

执行以上代码，输出结果为：

```
以下输出2017年3月份的日历:
     March 2017
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

>>>
```

calendar 模块包含的常用内建函数如表 4-3 所示。

表 4-3　calendar 模块的常用内建函数

序号	函数及描述
1	calendar.calendar(year,w=2,l=1,c=6) 返回一个多行字符串格式的 year 年年历，3 个月一行，间隔距离为 c。 每日宽度间隔为 w 字符。每行长度为 21×w +18+2×c。l 是每星期行数
2	calendar.firstweekday( ) 返回当前每周起始日期的设置。默认情况下，首次载入 calendar 模块时返回 0，即星期一

续表

序号	函数及描述
3	calendar.isleap(year) 是闰年返回 True，否则返回 False
4	calendar.leapdays(y1,y2) 返回在 y1，y2 两年之间的闰年总数
5	calendar.month(year,month,w=2,l=1) 返回一个多行字符串格式的 year 年 month 月日历，两行标题，一周一行。每日宽度间隔为 w 字符。每行的长度为 7×w+6。l 是每星期的行数
6	calendar.monthcalendar(year,month) 返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。year 年 month 月外的日期都设为 0；范围内的日子都由该月第几日表示，从 1 开始
7	calendar.monthrange(year,month) 返回两个整数。第一个是该月的星期几的日期码，第二个是该月的日期码。日从 0(星期一)到 6(星期日)；月从 1 到 12
8	calendar.prcal(year,w=2,l=1,c=6) 相当于 print calendar.calendar(year,w,l,c)
9	calendar.prmonth(year,month,w=2,l=1) 相当于 print calendar.calendar(year,w,l,c)
10	calendar.setfirstweekday(weekday) 设置每周的起始日期码。0(星期一)到 6(星期日)
11	calendar.timegm(tupletime) 与 time.gmtime 相反：接受一个时间元组形式，返回该时刻的时间戳(从 1970 起算经过的浮点秒数)
12	calendar.weekday(year,month,day) 返回给定日期的日期码。0(星期一)到 6(星期日)。月份为 1(一月) 到 12(十二月)

3. 获取随机数(random)模块

Python 中的 random 模块用于生成随机数。下面介绍一下 random 模块中最常用的几个函数。

(1) random.random。

random.random()用于生成一个 0 到 1 之间的随机浮点数 x：0≤x＜1.0。

(2) random.uniform。

random.uniform(a, b)用于生成一个指定范围内的随机浮点数，两个参数一个是上限，一个是下限。如果 a>b，则生成的随机数 n 的范围为 a≤n≤b；如果 a<b，则 b≤n≤a。

(3) random.randint。

random.randint(a, b)，用于生成一个指定范围内的整数。其中参数 a 是下限，b 是上限，下限必须不大于上限。生成的随机数 n 的范围为 a≤n≤b。

(4)　random.randrange。

random.randrange([start,] stop[, step])，从指定范围内，按指定基数递增的集合中获取一个随机数。如 random.randrange(20, 100, 2)，结果相当于从[20, 22, 24, 26, ..., 96, 98]序列中获取一个随机数。

(5)　random.choice。

random.choice 从序列中获取一个随机元素。函数原型为 random.choice(sequence)。参数 sequence 表示一个有序类型，可以是字符串、列表、元组等。

不难看出，random.randrange(20, 100, 2)在结果上与 random.choice(range(20, 100, 2)是等效的，都是随机产生 20~100 之间的一个偶数。

(6)　random.shuffle。

random.shuffle 的函数原型为 random.shuffle(x[, random])，用于将一个序列中的元素打乱。例如：

```
>>> L = ["Python", "is", "a","powerful,", "simple", "language"]
>>> random.shuffle(L)
>>> print(L)
['Python', 'is', 'language', 'powerful,', 'simple', 'a']
>>> random.shuffle(L)
>>> print(L)
['is', 'a', 'powerful,', 'simple', 'Python', 'language']
>>>
```

(7)　random.sample。

random.sample 的函数原型为 random.sample(sequence, k)，从指定序列中随机获取指定长度的片断，但不会修改原有序列。

需要说明的是，上面的 7 个函数都是随机函数，既然是“随机”的，就意味着每次返回的结果都可能不一样。在不同的机器上运行，结果更是如此。

4.4　巧用 lambda 表达式

Python 使用 lambda 来创建匿名函数。所谓匿名，意即不再使用 def 关键字以标准的形式定义一个函数。

lambda 表达式具有如下特点：

- lambda 只是一个表达式，函数体比 def 简单很多。
- lambda 的主体是一个表达式，而不是一个代码块，因而仅仅能在 lambda 表达式中封装有限的逻辑。
- lambda 函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。
- 虽然 lambda 函数看起来只能写一行，却不等同于 C 或 C++的内联函数，后者的目的是调用小函数时不占用栈内存从而提高运行效率。

语法：

lambda 函数的语法只包含一个语句，格式如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

【例 4-41】lambda 函数的使用——加法与减法：

```
#自定义函数
sum = lambda arg1, arg2: arg1 + arg2
sub = lambda arg1, arg2: arg1 - arg2
#调用 sum 函数
print ("相加的值: ", sum ( 10, 22 ))
print ("相减的值: ", sub ( 20, 5 ))
```

执行以上代码，输出结果为：

```
相加的值:  32
相减的值:  15
>>>
```

【例 4-42】lambda 函数的使用——立方与乘幂：

```
square = lambda x : x**3
print(square(3))             #27
power = lambda  x, y : x ** y
print(power(2, 10))           #1024
```

执行以上代码，输出结果为：

```
27
1024
>>>
```

【例 4-43】lambda 函数的使用——按姓氏排序：

```
names = ["Kitty Smit","Wart Kay","Jack Backus","Jim Gold"]
names.sort(key = lambda name:name.split()[-1])
print(",".join(names))
```

执行以上代码，输出结果为：

```
Jack Backus,Jim Gold,Wart Kay,Kitty Smit
>>>
```

注意：Python 在调用 lambda 表达式时绕过函数的栈分配。lambda 表达式运作起来就像一个函数，当被调用时，创建一个框架对象。

4.5　Python 工具箱

Python 提供了很多可供直接使用的文件包，前面介绍了 time 和 calendar 两个模块，下面再简要介绍一下 os 和 file 两个模块，详细的应用参见第 5 章。

1. os(操作系统)模块

os 模块提供了非常丰富的方法用来处理文件和目录，使用时，一定要加上导入语句 import os。

os 模块常用的内建函数如表 4-4 所示。

表 4-4　os 模块的常用内建函数

序号	函数及描述
1	os.access(path, mode) 检验权限模式
2	os.chdir(path) 改变当前工作目录
3	os.chflags(path, flags) 设置路径的标记为数字标记
4	os.chmod(path, mode) 更改权限
5	os.chown(path, uid, gid) 更改文件所有者
6	os.chroot(path) 改变当前进程的根目录
7	os.close(fd) 关闭文件描述符 fd
8	os.lchmod(path, mode) 修改连接文件权限
9	os.dup(fd) 复制文件描述符 fd
10	os.dup2(fd, fd2) 将一个文件描述符 fd 复制到另一个 fd2
11	os.fchdir(fd) 通过文件描述符改变当前工作目录
12	os.fchmod(fd, mode) 改变一个文件的访问权限，该文件由参数 fd 指定，参数 mode 是 Unix 下的文件访问权限
13	os.fchown(fd, uid, gid) 修改一个文件的所有权，这个函数修改一个文件的用户 ID 和用户组 ID，该文件由文件描述符 fd 指定
14	os.fdatasync(fd) 强制将文件写入磁盘，该文件由文件描述符 fd 指定，但是不强制更新文件的状态信息
15	os.fdopen(fd[, mode[, bufsize]]) 通过文件描述符 fd 创建一个文件对象，并返回这个文件对象
16	os.fpathconf(fd, name) 返回一个打开的文件的系统配置信息。name 为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定(POSIX.1、Unix 95、Unix 98 和其他)

续表

序号	函数及描述
17	os.fstat(fd) 返回文件描述符 fd 的状态，类似于 stat()
18	os.fstatvfs(fd) 返回包含文件描述符 fd 的文件系统的信息，类似于 statvfs()
19	os.fsync(fd) 强制将文件描述符为 fd 的文件写入硬盘
20	os.ftruncate(fd, length) 裁剪文件描述符 fd 对应的文件，所以它最大不能超过文件大小

2. file(文件)模块

file 对象使用 open()函数来创建，可以直接使用，不需要导入。file 模块常用的内建函数如表 4-5 所示。

表 4-5　file 模块的常用内建函数

序号	函数及描述
1	file.close() 关闭文件。关闭后文件不能再进行读写操作
2	file.flush() 刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件，而不是被动地等待输出缓冲区写入
3	file.fileno() 返回一个整型的文件描述符(File Descriptor，FD)，可以用在如 os 模块的 read 方法等一些底层操作上
4	file.isatty() 如果文件连接到一个终端设备，则返回 True，否则返回 False
5	file.next() 返回文件下一行
6	file.read([size]) 从文件读取指定的字节数，如果未给定或为负，则读取所有字节
7	file.readline([size]) 读取整行，包括“\n”字符
8	file.readlines([sizehint]) 读取所有行并返回列表，若给定 sizehint>0，返回总和大约为 sizehint 字节的行，实际读取值可能比 sizehint 较大，因为需要填充缓冲区
9	file.seek(offset[, whence]) 设置文件当前位置
10	file.tell() 返回文件当前位置

3. 一些有用的方法或功能

在实际编写程序的过程中，经常会用到一些常用的方法或功能，如表 4-6 所示。

表 4-6　常用方法或功能

方法或功能	方法或功能的说明
locals()	返回当前变量作用域中的变量集合
“+”操作符	用于字符串时将连接两个字符串，用于数值时将两个数相加
with 关键字	可用于处理打开文件的关闭工作，也可与 as 关键字结合使用
sys.stdout	Python 中所谓的“标准输出”，可以从标准块 sys 模块访问
pickle 模块	容易而高效地将 Python 数据对象保存到磁盘以及从磁盘恢复
help()	允许在 IDLE Shell 中访问 Python 文档
find()	在一个字符串中查找特定子串
setup.py 程序	提供模块的元数据，用来构建、安装和上传打包的发布
len()	提供某个数据对象的长度，或者统计一个集合的项数，如列表中的项数

4.6　案例实训：“哥德巴赫猜想”的验证

哥德巴赫猜想是公认的世界数学难题，我国著名数学家陈景润院士对此做了毕生的研究。任何一个大于 2 的偶数都可以分解为两个素数之和，这就是著名的哥达巴赫猜想，简称 1+1(陈景润已证明了 1+2)。本例运行时，要求输入一个大于 2 的偶数，程序运行后，输出两个素数，其和正好等于该偶数。

程序如下：

```
#导入数学模块
import math
#判断是否为素数
def is primer(num):
    flag = 1
    if num==1 or num==2:
        flag = 1
    else:
        end=int(math.sqrt(num))
        #循环次数为该数的平方根取整
        for j in range(2,end+1):
            #余数为 0，除尽，不是素数
            if num%j==0:
                flag = 0
    return flag
#判断哥德巴赫猜想是否成立
def is gdbh(num):
    flag1 = 0
    if num % 2 == 0 and num > 2:
        #循环次数为偶数的一半
```

```
        for j in range(1,num//2+1):
            #判断由偶数拆分成的两个数是否均为素数
            bl1 =is primer(j)
            bl2 =is primer(num-j)
            if bl1==1 and bl2==1:
                print("{0}={1}+{2}".format(num, j, num - j))
                flag1 = 1
                break
    return flag1
#测试函数
def test():
    print("输入一个大于 2 的偶数：")
    x=int(input())
    while x<=2 or x%2==1:
        print("输入一个大于 2 的偶数：")
        x=int(input())
    if is gdbh(x)==1:
        print("{0}能写成两个素数的和,符合哥德巴赫猜想。".format(x))
#执行测试函数
if   name    == "  main  ":
        test()
```

执行以上代码，输出结果为：

```
输入一个大于2的偶数：
385498
385498=5+385493
385498能写成两个素数的和,符合哥德巴赫猜想。
>>>
```

4.7 本 章 小 结

本章介绍了与函数相关的内容。首先介绍了 Python 代码编写规范、代码的风格，并举例加以说明；然后介绍了自定义函数和自定义模块两部分，包括函数的定义、递归、变量作用域、迭代器、生成器等；随后介绍了模块导入语句的使用、Python 常用的内建函数(如输入和输出函数等)的使用，尤其是详细地介绍了格式化输出的方法；介绍了 lambda 表达式语法格式及其使用方法；最后介绍了 os 模块、file 模块以及其他常用的方法或功能。

习 题

1. 填空题

(1) 函数定义以关键字(　　　)开始，该行最后以(　　　)结束。

(2) 没有 return 语句的函数将返回(　　　)。

(3) 函数定义时声明的参数称为(　　　)，而函数调用时提供的参数称为(　　　)。

(4) 使用关键字(　　　)可以在一个函数中设置一个全局变量。

(5) 设有 f=lambda x, y :{x:y}，则 f(2,3)的值是(　　　)。

(6) Python 包含了数量众多的模块，通过(　　　)语句可以导入模块，并使用其定义的功能。

(7) 设 Python 中有模块 m，如果希望同时导入 m 中的所有成员，则可以采用(　　　)的导入形式。

(8) 建立模块 big.py，模块内容如下：

```
def B () :
   print ('Python')
def A () :
   print ('hello')
```

为了调用模块中的 A()函数，应先使用语句(　　　)。

2. 选择题

(1) 下列选项中不属于函数优点的是(　　)。

A. 减少代码重复　　B. 使程序模块化

C. 使程序便于阅读　　D. 便于发挥程序员的创造力

(2) 以下关于函数的说法正确的是(　　)。

A. 函数定义时必须有形参

B. 函数中定义的变量只在该函数体中起作用

C. 函数定义时必须带 return 语句

D. 实参与形参的个数可以不相同、类型可以任意

(3) 以下关于函数的说法正确的是(　　)。

A. 函数的实际参数和形式参数必须同名

B. 函数的形式参数既可以是变量也可以是常量

C. 函数的实际参数不可以是表达式

D. 函数的实际参数可以是其他函数的调用

(4) 有以下两个程序。

程序 1:

```
a=[1,2,3,4,5]
def  f(a):
     a=a+[6]
f(a)
print(a)
```

程序 2:

```
b=[1,2,3,4,5]
def  f(b):
      b+=[6]
f(b)
print(b)
```

下列说法正确的是(　　)。

A. 两个程序均能正确运行，但结果不同

B. 两个程序的运行结果相同

C. 程序 1 能正确运行，程序 2 不能

D. 程序 1 不能正确运行，程序 2 能

(5) 已知 f=lambda a, b:a+b，则 f([4],[1,2,3,5])的值是(　　)。

A. [1,2,3,5,4]　　B. 15　　C. [4,1,2,3,5]　　D. {1,2,3,4,5}

(6) 下列语句的运行结果是(　　)。

```
f1=lambda a:a*3
f2=lambda a:a**3
print(f1(f2(4)))
```

A. 106　　B. 148　　C. 136　　D. 192

(7) 下列程序执行后，w 的值是(　　)。

```
def f(a,b):
   return a**3+b**2
w=f(f(1,2),5)
print(w)
```

A. 100　　B. 150　　C. 35　　D. 9

3. 问答题

(1) 简单叙述 Python 函数参数的类型。

(2) 什么是 lambda 函数？它有什么用处？

(3) 如何在一个 function 里面设置一个全局变量？

(4) Python 是如何进行类型转换的？

(5) Python 是如何进行内存管理的？

4. 实验操作题

(1) 编程输出斐波那契数列的前若干项。即根据用户输入的正整数，输出数列的各项，如输入正整数 5，则输出斐波那契数列的前五项：1，1，2，3，5。

(2) 用函数实现最大公约数算法和最小公倍数算法，编写测试程序测试这两个算法。

(3) 编程获取五天前的年月日。

(4) 编程计算某几个月的天数。

(5) 编程将字符串转换为大写字母，或者将字符串转为小写字母。

(6) 使用函数库完成十进制数转二进制、八进制、十六进制数的运算。



第 5 章

文件与异常处理

本章要点

(1) 文件和文件对象。
(2) 文本文件的读写。
(3) os 模块的文件操作方法。
(4) shutil 模块的文件操作方法。
(5) CSV、Excel 文件的基本操作。
(6) HTML、XML 文档的基本操作。
(7) Python 异常处理机制。

学习目标

(1) 掌握文件的基本操作以及对目录的操作方法。
(2) 掌握 CSV、Excel 文件的读写方法。
(3) 掌握 HTML、XML 文档的操作方法。
(4) 理解 Python 的异常处理机制。
(5) 掌握捕捉异常的基本方法。

之前介绍的 input、print 函数是与外部交互的函数，它们是对标准输入输出设备(即键盘和屏幕)进行操作的。除此而外，Python 还可以对文件进行操作，实现更多的外部交互。本章 5.1 节介绍打开文件、关闭文件、创建文件等基本操作。在对文件和文件夹进行操作时会用到 os 模块和 shutil 模块，本章 5.2 节对此予以介绍。除一般文件外，Python 还可以对 CSV、Excel、HTML、XML 等文件进行创建、读写等操作，5.3~5.6 节将做具体介绍。

在编程过程中，难免会出现异常情况，例如用一个数除以零就是异常，我们是不希望出现这种异常错误的，所以在编程时，需要判断正常和异常情况，并对异常情况进行处理，以免发生错误，影响程序运行。5.7 节和 5.8 节将对此做详细介绍。

5.1 文件的基本操作

5.1.1 打开文件

使用文件之前，须首先打开文件，然后进行读、写、添加等操作。Python 打开文件使用 open 函数，其语法格式为：

```
open(name[,mode[,buffering]])
```

其中，文件名 name 为必选参数，模式 mode 和缓冲 buffering 是可选参数。该函数返回一个文件对象。

【例 5-1】打开一个文本文件：

```
f = open(r"C:\Users\test.txt")
```

上述语句直接打开一个指定的文件，如果文件不存在则创建该文件。这里的 f 是一个文件对象，它与指定的文件建立了关联，很多文献称 f 为文件描述符，实际上它可视为指

定文件的“句柄”，所有对指定文件的后续操作都将通过这个句柄进行，直到使用后面将要介绍的 close()函数关闭指定文件为止。

如果 open 函数后面的参数中只带一个文件名，我们只是获得了能读取文件内容的文件对象(即上面的 f)。若要在文件中写入内容，就必须要提供一个模式参数来显式地声明。open 函数中的模式参数如表 5-1 所示。

表 5-1　open 函数中的模式参数

参 数 值	描　述
‘r’	读模式
‘w’	写模式
‘a’	追加模式
‘b’	二进制模式(可添加到其他模式中使用)
‘+’	读/写模式(可添加到其他模式中使用)

其中读模式是默认模式。写模式即向文件中写入内容。‘+’参数可以用到其他任何模式中，指明读和写都是允许的。‘b’模式用于改变处理文件的方法。一般来说，Python 处理文本文件(包括字符)时没有问题，但如果处理的是一些其他类型的文件(二进制文件)，如声音剪辑或图像等，则应在模式参数中增加‘b’，明确指出按二进制形式来处理文件。

模式参数组合及描述如表 5-2 所示。

表 5-2　模式参数组合及其描述

模式参数组合	描　述
r+	以读写模式打开
w+	以读写模式打开(参见 w)
a+	以读写模式打开(参见 a)
rb	以二进制读模式打开
wb	以二进制写模式打开(参见 w)
ab	以二进制追加模式打开(参见 a)
rb+	以二进制读写模式打开(参见 r+)
wb+	以二进制读写模式打开(参见 w+)
ab+	以二进制读写模式打开(参见 a+)

这里需要强调一下使用二进制模式的理由。使用二进制模式读写文件时，与使用文本模式不会有很大区别。但是，在使用二进制模式时，Python 会原样给出文件中的内容，在文本模式下则不一定。Python 对于文本文件的操作方式中唯一要用到的技巧是标准化换行符。一般来说，换行符(\n)表示结束一行并另起一行，这也是 Unix 系统中的规范，但在 Windows 中一行结束的标志是\r\n。为使程序能跨平台运行，Python 在这里做了一些自动转换：当在 Windows 下用文本模式读文件中的文本时，Python 将\r\n 转换成\n；相反地，在 Windows 下用文本模式向文件中写文本时，Python 将\n 转换成\r\n。

在使用二进制文件时可能会出现问题，因为文件中可能包含被解释成换行符的字节，而使用文本模式时会自动转换。这样会破坏二进制数据，所以使用二进制模式时，不会发生转换。

注意： 通过在模式参数中使用 U 参数，能在打开文件时使用通用的换行符支持模式，在这种模式下，所有的换行符(\r\n、\r 或\n)都被转换成\n，而不考虑所运行的平台。

open 函数的第三个参数控制文件的缓冲，对参数值的说明如表 5-3 所示。

表 5-3　open 函数的缓冲参数

参 数 值	描　述
0(False)	I/O 无缓冲，即所有读写操作直接针对硬盘
1(True)	I/O 有缓冲，即使用内存代替硬盘
>1	大于 1 的数字表示缓冲区的大小(以字节为单位)
−1(或任何负数)	表示使用默认的缓冲区大小

5.1.2　关闭文件

文件使用完毕后应及时关闭。在 Python 中关闭文件用 close 方法。通常来说，Python 会在一个文件不用后自动将其关闭，不过这一功能没有保证，因为 Python 可能会缓存写入的数据，如果程序因为某种原因崩溃，数据就有可能没有完整地写入到文件中，从而引发文件故障。因此，最好还是养成自己关闭文件的习惯。如果一个文件在关闭后还对其进行操作，则会产生 ValueError 错误。

【例 5-2】关闭文本文件。

要关闭例 5-1 中的 f 文件对象，可以使用如下语句：

```
f.close()
```

该语句执行后，f 与 test.txt 的关联不复存在，当然也就不能再对 test.txt 文件进行读写了，除非再度打开。

5.1.3　在文本文件中读取数据

在文本文件中读取数据的语法格式为：

```
f.read([size])          #size 为读取的长度，以 byte 为单位
f.readline([size])       #读一行，如果定义了 size，有可能返回的只是一行的一部分
f.readlines([size])      #把文件每一行作为 list 的一个成员，并返回这个 list
```

其实，readlines()的内部是通过循环调用 readline()来实现的。如果提供 size 参数(size 是表示读取内容的总长)，则可能只读到文件的一部分。

【例 5-3】读取文本文件内容：

```
>>> f = open(r"C:\Users\test.txt")
>>> f.read(5)
'Hello'
>>> f.close()
>>> f = open(r"C:\Users\test.txt")
>>> f.readline()
'Hello World!'
>>> f.close()
```

这里假设在 C:\Users 目录下有一个文本文件 test.txt，文本内容为“Hello World!”。
可以看出，readline()将文件对象 f(也就是文本文件 test.txt)的一行内容读出来了。

5.1.4　创建文本文件

在 Python 中，以追加模式打开文本文件即可创建此文件，语法格式为：

```
open(name, 'a' [,buffering])          #创建空文件
```

【例 5-4】创建文本文件。

在 C 盘的 Users 目录下创建一个文本文件 text. txt，可使用语句：

```
f = open(r'C:\Users\text.txt','a')
```

语句执行后，将在相应的目录下生成一个名为 text.txt 的文件。因尚未向其中添加数据，其字节数为 0。

5.1.5　向文本文件中添加数据

向文件中写入数据的函数是 write()和 writelines()，其语法格式为：

```
f.write(str)           #把 str 写到文件中，write()并不会在 str 后加上一个换行符
f.writelines(seq)       #把 seq 的内容全部写到文件中(多行一次性写入)。
                         #这个函数也只是忠实地写入，不会在每行后面加上任何东西
```

【例 5-5】向文本文件中添加数据。

假设我们向 D:\xunlian\test.txt 文件中写入数据，可以使用下列语句：

```
>>> f = open(r"D:\xunlian\test.txt",'w')
>>> str = 'Welcome to China!'
>>> f.write(str)
17
>>> f.close()
>>> f = open(r"D:\xunlian\test.txt",'r')
>>> f.read()
'Welcome to China!'
>>> f.close()
```

其中的 17 表示向文本文件 test.txt 中写入了 17 个字符。

5.1.6　文件指针

假设我们读取文本文件 test.txt，文本内容为“Welcome to China!”，当用两个 read 方法读取时，第一个 read 返回‘Welcome to China!’，第二个 read 返回‘’，即不能重复读取，这是为什么呢？

这种现象与文件指针有关。对文件操作时，文件内部会有一个文件指针来定位当前位置，控制文件指针位置可以实现重复读取，用 seek 方法可以控制文件指针的位置，其语法格式为：

```
seek(offset[, whence])              #移动文件指针
```

各参数的含义如下：

offset：偏移量。一般是相对于文件的开头来计算的，且一般为正数。

whence：偏移相对位置。whence 可以为 0，表示从头开始计算，为 1 则表示以当前位置为原点计算，为 2 则表示以文件末尾为原点进行计算。

注意： 如果文件以 a 或 a+的模式打开，每次进行写操作时，文件操作标记都会自动返回到文件末尾。

偏移相对位置常量有 SEEK_SET、SEEK_CUR、SEEK_END。

os.SEEK_SET：表示文件的起始位置，即 0(默认情况)，此时 offset 必须为 0 或正数。

os.SEEK_CUR：表示文件的当前位置，即 1，此时 offset 可以为负数。

os.SEEK_END：表示文件的结束位置，即 2，此时 offset 通常为负数。

欲获取文件指针位置，可以使用 tell 方法，其语法格式为：

```
f.tell()              #返回文件操作标记的当前位置，以文件的开头为原点
```

【例 5-6】获取文件指针的当前位置。

上一例中，test.txt 文件中的文本内容为“Welcome to China!”，若第二次读取，则会输出‘’，可以使用 seek 函数使其从头开始读取：

```
>>> f = open(r"D:\xunlian\test.txt",'r')
>>> f.readline()
'Welcome to China!'
>>> f.seek(0)
0
>>> f.readline()
'Welcome to China!'
>>> f.tell()
17
>>> f.close()
```

可见，从文件中读出“Welcome to China!”后，文件指针的当前位置为 17。

5.1.7　截断文件

截断文件使用 truncate 方法，把文件截成规定的大小，默认的是截到当前文件操作标记的位置。截断文件的语法格式为：

```
f.truncate([size])
```

如果 size 比文件的大小还要大，依据系统的不同，可能是不改变文件，也可能是用 0 把文件补到相应的大小，也可能是把一些随机的内容加上去。

【例 5-7】截断文件。

在 test.txt 文件中又写入一行“Thank you very much!”，看截断后能否再输出：

```
>>> f = open(r"D:\xunlian\test.txt",'r+')
>>> f.truncate(18)
18
>>> f.readline()
'Welcome to China! '
>>> f.readline()
''
>>> f.close()
```

可以看出，截断后读出的内容为空串，即第 18 个字符以后的内容读不出来，亦即截断后不能再输出。

5.1.8　复制、删除、移动、重命名文件

复制文件使用 shutil 模块中的方法，涉及到的方法(函数)有 copy、copyfile、copytree。下面分别对各个方法进行说明：

```
shutil.copy(src, dst)        #复制数据从 src 到 dst(src 为文件，dst 可以为目录)
shutil.copyfile(src, dst)    #复制数据从 src 到 dst(src 和 dst 均为文件)
shutil.copytree(src, dst)    #递归复制文件夹，其中 src 和 dst 均为目录，且 dst 不存在
```

删除文件使用 os 模块中的 remove 方法：

```
os.remove(path)                #path 为文件的路径名
```

移动文件使用 shutil 模块中的 move 方法：

```
shutil.move(src, dst)  #移动数据从 src 到 dst，src 和 dst 可以为文件，也可以为目录
```

重命名文件或目录使用 os 模块中的 rename 方法：

```
os.rename(old, new)            #old 为原文件名，new 为更改后的文件名
```

【例 5-8】使用 copy 方法复制文件：

```
>>> import shutil
>>> import os
>>> os.chdir(r'D:')
>>> shutil.copy(r'D:\xunlian\test1.txt','D:\practice')
'D:\\practice\\test1.txt'
>>> shutil.copy(r'D:\practice\test1.txt',r'D:\practice\test2.txt')
'D:\\practice\\test2.txt'
```

第一个 shutil.copy()将 D:\xunlian 下的 test1.txt 文件复制到 D:\practice 文件夹下；第二个 shutil.copy()将 D:\practice 下的 test1.txt 文件复制到此文件夹下，命名为 test2.txt。

在 copy 方法中，如果 dst 是文件夹，则把 src 文件复制到该文件夹中；如果 dst 是文件，则把 src 文件复制到 dst 文件中，即复制+重命名。本例及后面部分例题中用到的 os.chdir()是 os 模块中切换到指定目录所用的方法。

【例 5-9】使用 copyfile 方法复制文件。

使用 copyfile 方法的前提是目标文件具有写权限，否则将会产生 IoError 错误。我们使用 glob(pathname)函数返回所有匹配的文件路径列表，这里既可以是绝对路径，也可以是相对路径。

```
>>> import shutil
>>> import glob
>>> import os
>>> os.chdir(r'D:\practice')
>>> print('before:',glob.glob('list.*'))
before: ['list.txt']
>>> shutil.copyfile('list.txt','list.txt.copy')
'list.txt.copy'
>>> print('after:',glob.glob('list.*'))
after: ['list.txt', 'list.txt.copy']
```

可以看到，shutil.copyfile()将 D:\practice 文件夹下的 list.txt 复制并命名为 list.txt.copy。

【例 5-10】使用 copytree 方法复制文件：

```
>>> import os
>>> import shutil
>>> import tempfile
>>> dir1 = tempfile.mktemp('.dir')  #返回一个临时文件的路径，但不创建该临时文
件
>>> os.mkdir(dir1)
>>> dir2 = dir1 + '.copy'
>>> print(dir1,dir2)
C:\Users\BBQ\AppData\Local\Temp\tmppgkpkj16.dir C:\Users\BBQ\AppData\Local\
Temp\tmppgkpkj16.dir.copy
>>> shutil.copytree(dir1,dir2)
'C:\\Users\\BBQ\\AppData\\Local\\Temp\\tmppgkpkj16.dir.copy'
```

shutil.copytree()将创建的临时文件 dir1 复制到 dir2，即 dir1.copy。

【例 5-11】文件删除。

使用 remove()方法删除 D:\practice 目录下的 text.txt 文件：

```
import os
os.chdir(r'D:\practice')
os.remove('text.txt')
```

执行上述命令后，D:\practice 目录下的 text. txt 文件不复存在。

【例 5-12】文件移动。

使用 move 方法将文件或文件夹移动到另一目录，使用 glob 函数获得文件路径：

```
>>> import shutil
>>> import glob
>>> import os
>>> os.chdir(r'D:\practice')
>>> print('before:',glob.glob('new.*'))
before: ['new.txt']
>>> shutil.move('new.txt','new.out')
'new.out'
>>> print('after:',glob.glob('new.*'))
after: ['new.out']
```

shutil.move()将 D:\practice 目录下的 new.txt 移动到当前目录下，命名为 new.out。

【例 5-13】文件重命名。

把当前目录下的文件 text. txt 重命名为 text1. txt，使用的语句为：

```
os.rename('text.txt','text1.txt')
```

5.2　指定目录下的文件操作

5.2.1　获取当前目录

获取 Python 当前脚本运行目录的方法为 getcwd()，其语法格式为：



```
os.getcwd()
```

【例 5-14】得到当前工作空间的目录：

```
>>> import os
>>> f = os.getcwd()
>>> f
'D:\\practice'
```

5.2.2　获取当前目录下的内容

os 模块下的 listdir 方法用于获取当前目录下所有的文件和目录名，其语法格式为：

```
os.listdir()
```

【例 5-15】获取指定文件夹下面的所有文件及文件夹，如果指定的文件夹不存在，则返回相应的提示信息：

```
import os
def listdir(dir path):
    if os.path.exists(dir path):
        return os.listdir(dir path)
    else:
        return '目录'+ dir path + '不存在'

if   name   == "  main  ":
    f=listdir(r"d:\practice")    #该目录存在
    print(f)
    f=listdir(r"d:\practices")   #该目录不存在
    print(f)
```

```
['list.txt', 'list.txt.copy', 'new.out', 'test1.txt', 'test2.txt',
'text1.txt']
目录D:\practices不存在
>>>
```

5.2.3　创建、删除目录

创建单个目录的语法格式为：

```
os.mkdir("file")
```

删除目录有两种方法，分别调用 os 模块的 rmdir 方法和 shutil 的 rmtree 方法，不同的是前者只能删除空目录，而后者空目录和非空目录均可删除。

```
os.rmdir("dir")                         #只能删除空目录
shutil.rmtree("dir")                       #空目录、有内容的目录都可以删
```

【例 5-16】创建新目录：

```
import os
os.mkdir(r'D:\newdir')
```

【例 5-17】删除空目录，首先判断是否是空目录：

```
import os
def delete_dir(dir):
if os.path.isdir(dir):
    for item in os.listdir(dir):
      if item!='System Volume Information':
         delete_dir(os.path.join(dir, item))
      if not os.listdir(dir):
         os.rmdir(dir)

f = delete_dir(r'D:\newdir')
```

运行上面的代码，将删除 D 盘下的 newdir 目录(前提是目录为空)。

【例 5-18】使用 rmtree 方法删除目录：

```
import shutil
dir_path = r'D:\test'
shutil.rmtree(dir_path)
```

5.3 CSV 文件

CSV 是逗号分隔值 Comma-Separated Values 的缩写，其文件以纯文本形式存储表格数据(数字和文本)。CSV 并不是一种单一的、定义明确的格式(尽管 RFC 4180 有一个被通常使用的定义)。因此在实践中，术语 CSV 泛指具有以下特征的任何文件。

(1) 纯文本，使用某个字符集，比如 ASCII、Unicode、EBCDIC 或 GB2312。

(2) 由记录组成(典型的是每行一条记录)。

(3) 每条记录被分隔符分隔为字段(典型分隔符有逗号、分号或制表符；有时分隔符可以包括可选的空格)。

(4) 每条记录都有同样的字段序列。

通常可以使用 Wordpad、Notepad(记事本)或 Excel 来打开 CSV 文件。

一般情况下，我们用 Excel 生成的文件的扩展名是 xls 或 xlsx，如果直接重命名为 CSV 扩展名(下称 CSV 格式)，会报错。其解决方法是，将 Excel 生成的表直接存为 CSV 格式，或将原有文件另存为 CSV 格式。

Python 本身就带有 CSV 包，使用时，先用 import csv 导入即可，使用 CSV 包可以对 CSV 文件进行读写操作。

5.3.1 读 CSV 文件

在 Python 的 csv 模块中，有读 CSV 文件的方法 reader()，下面举例说明如何使用 reader 函数读取 CSV 文件。

【例 5-19】CSV 文件的读取。

有如下 CSV 文件：

80082	4432	4355	2345
9888.43	4325.6	89331	435
43772.9	477	9334	325

读取 CSV 数据的代码及运行结果为：

```
>>> import csv
>>> csv_reader = csv.reader(open(r'D:\xunlian\one.csv',encoding='u
tf-8'))
>>> for row in csv_reader:
        print(row)


['80082\t4432\t4355\t2345']
['9888.43\t4325.6\t89331\t435']
['43772.9\t477\t9334\t325']
```

5.3.2　写 CSV 文件

在 Python 中，写 CSV 文件使用 csv 模块中的 writer 方法和 writerow 方法。writer 方法用于将数据转化为带分隔符的字符串(给定文件对象的模式必须为‘w’)，writerow 方法用于将一行数据写入文件中。

【例 5-20】在上述 CSV 文件中写入数据“1，2，3，4”：

```
import csv
list = ['1','2','3','4']
out = open(r'D:\xunlian\one.csv','w')
csv writer = csv.writer(out)
csv writer.writerow(list)
out.close()
```

直接使用这种写法可能会导致文件每一行后面多一个空行。解决方案如下：

```
out = open(r'D:\xunlian\one.csv','w',newline='')
csv writer = csv.writer(out, dialect='excel')
csv writer.writerow(list)
out.close()
```

将上述代码放入一个程序中，运行后，可以得到如图 5-1 所示的 one.csv 文件。

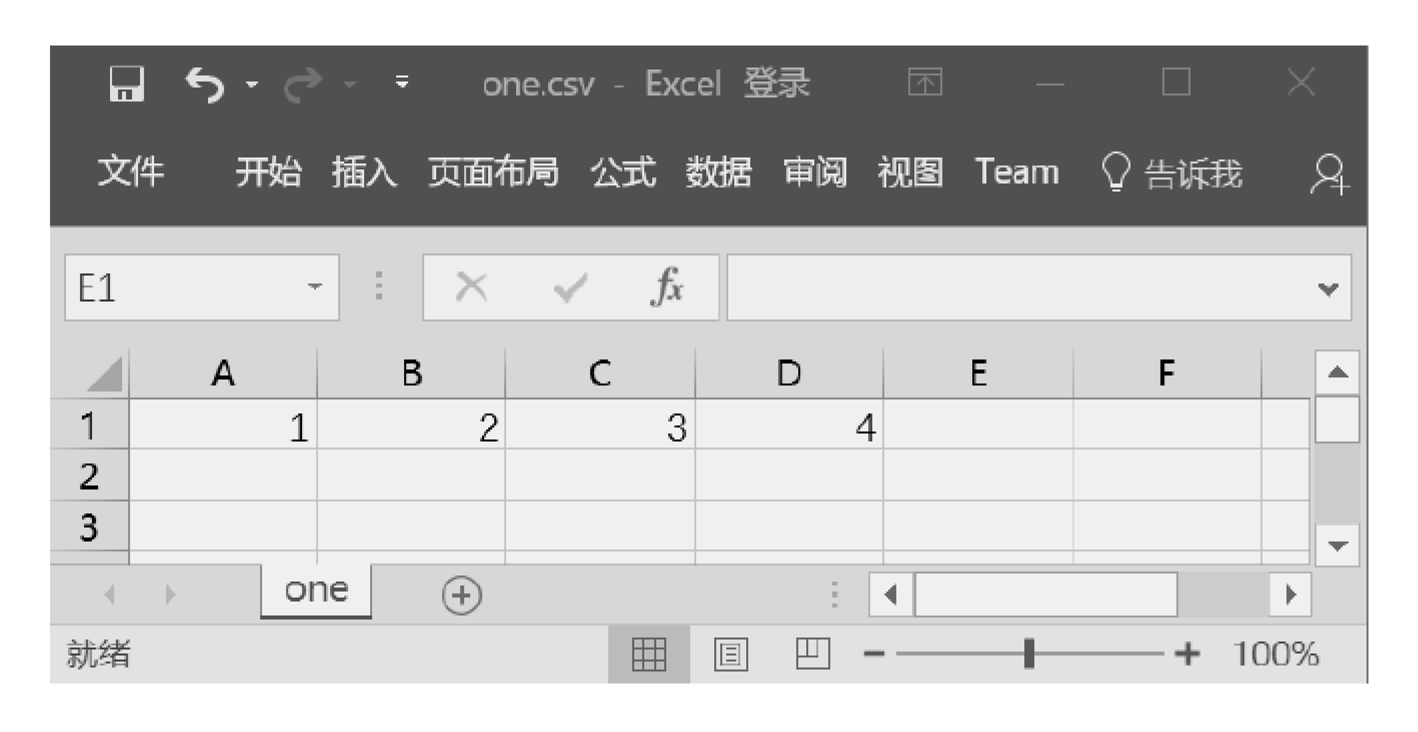


图 5-1　例 5-20 的运行结果

5.4　Excel 文件

在学习和工作的过程中，我们经常用到 Excel 文件，Python 也可以处理 Excel 文件。操作 Excel 文件主要用到 xlrd 和 xlwt 两个库，xlrd 是读 Excel 文件的模块，xlwt 是写 Excel 文件的模块。在对 Excel 文件进行读写操作时，需要先下载并安装 xlrd 库和 xlwt 库。

5.4.1　使用 xlrd 读 Excel 文件

xlrd 提供的接口较多，常用的是：

```
open_workbook()                #打开指定的 Excel 文件，返回一个 Book 对象
```

通过 Book 对象，可以得到各个 Sheet 对象(一个 Excel 文件可以有多个 Sheet，每个 Sheet 就是一张表格)。例如：

```
Book.nsheets                   #返回 Sheet 的数目
Book.sheets()                  #返回所有 Sheet 对象的 list

Book.sheet_by_index(index)
  #返回指定索引处的 Sheet。相当于 Book.sheets()[index]

Book.sheet_names()              #返回所有 Sheet 对象名字的 list
Book.sheet_by_name(name)        #根据指定 Sheet 对象名字返回 Sheet
```

通过 Sheet 对象可以获取各个单元格，每个单元格是一个 Cell 对象：

```
Sheet.name                     #返回表格的名称
Sheet.nrows                    #返回表格的行数
Sheet.ncols                    #返回表格的列数
Sheet.row(r)                   #获取指定行，返回 Cell 对象的 list
Sheet.row_values(r)              #获取指定行的值，返回 list
Sheet.col(c)                   #获取指定列，返回 Cell 对象的 list
Sheet.col_values(c)              #获取指定列的值，返回 list
Sheet.cell(r, c)                #根据位置获取 Cell 对象
Sheet.cell_value(r, c)            #根据位置获取 Cell 对象的值
Cell.value                     #返回单元格的值
```

【例 5-21】使用 xrld 读 Excel 文件。

test.xls 文档的内容如下：

1	2	3	4	5	6
a	b	c	d	e	f
7	8	9	0	1	2
g	h	i	j	k	l

使用 xlrd 读取此文档的代码为：

```
import xlrd
wb = xlrd.open_workbook(r'D:\xunlian\test.xls')
#打印每张表的最后一列
#方法 1
```

```
for s in wb.sheets():
    print("The last column of sheet %s:" %(s.name))
    for i in range(s.nrows) :
        print(s.row(i)[-1].value)
#方法 2
for i in range(wb.nsheets):
    s = wb.sheet by index(i)
    print("The last column of sheet %s:" %(s.name))
    for v in s.col values(s.ncols - 1) :
        print(v)
#方法 3
for name in wb.sheet names( ):
    print("The last column of sheet %s:" % (name))
    s = wb.sheet by name(name)
    c = s.ncols -1
    for r in range(s.nrows) :
        print(s.cell_value(r, c))
```

将上述代码放入一个程序中，运行结果为：

```
The last column of sheet Sheet1:
6.0
f
2.0
1
The last column of sheet Sheet1:
6.0
f
2.0
1
The last column of sheet Sheet1:
6.0
f
2.0
1
```

5.4.2　使用 xlwt 写 Excel 文件

xlwt 提供的接口相对 xlrd 来说要少，主要有：

```
Workbook()                    #构造函数，返回一个工作簿的对象
Workbook.add sheet(name)        #添加了一个名为 name 的表，类型为 Worksheet

Workbook.get sheet(index)
  #可以根据索引返回 Worksheet(前提是已经添加到 Workbook 中了)

Worksheet.write(r, c, value)     #将 value 填充到指定位置
Worksheet.row(n)               #返回指定的行
Row.write(c, value)             #在某一行的指定列写入 value
Worksheet.col(n)               #返回指定的列
#通过对 Row.height 或 Column.width 赋值，可以改变行或列默认的高度或宽度
#(单位：0.05 pt，即 1/20 pt)
Workbook.save(filename)        #保存文件
```

注意：

- xlwt 模块至多能写 65535 行、256 列，如果超过这个范围，程序运行就会出现错误，这时，需要通过其他一些途径来解决。如果我们只注重数据的处理，那么可

以采用 csv 模块来替代。

- 文件默认的编码方式是 ASCII，要改变编码方式，指定 Workbook()的 encoding 参数即可，如 Workbook(encoding='utf-8')。
- 表的单元格默认是不可重复写的，如果有需要，在调用 add_sheet()的时候指定参数 cell_overwrite_ok=True 即可。

【例 5-22】向新的 Excel 文件中写入数据并保存文件：

```
import xlwt
book = xlwt.Workbook(encoding='utf-8')
sheet = book.add sheet('sheet test', cell overwrite ok=True)
sheet.write (0, 0, 'mike')
sheet.row(0).write(1, '&')
sheet.write (0, 2, 'jack')
sheet.col(2).width = 300
book.save(r'D:\xunlian\test1.xls')
```

将上述代码放入一个程序中，运行后，可以得到如图 5-2 所示的 test1.xls 文件。

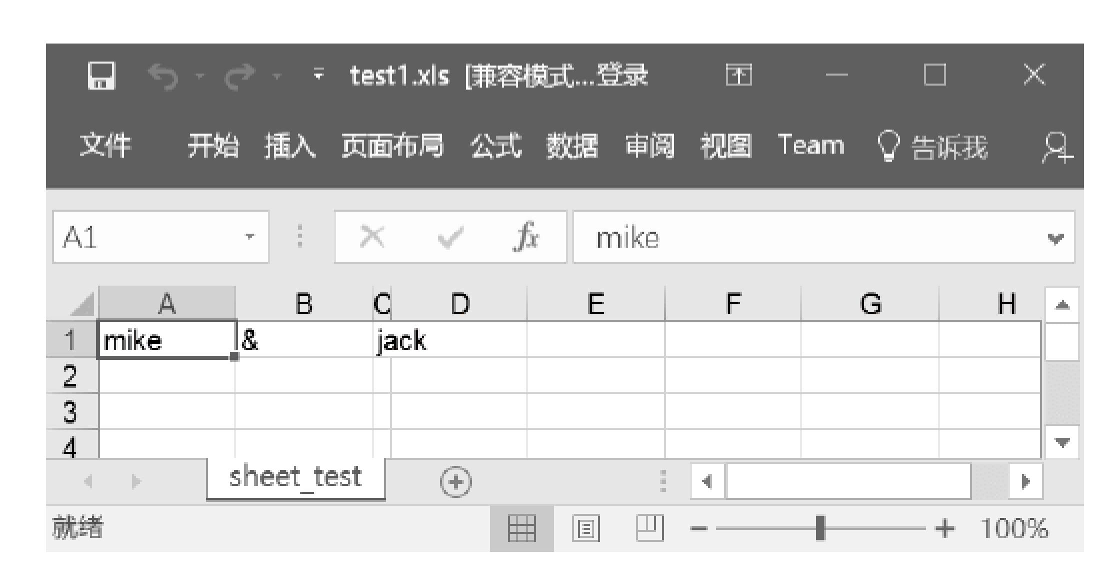


图 5-2　例 5-22 的运行结果

除了写入数据，xlwt 还可以改变单元格格式。write 方法允许接受一个 XFStyle(Excel File Style)类型的参数，置于最后。

使用 easyxf()可快速生成一个 XFStyle 对象。

【例 5-23】使用 xlwt 改变单元格格式：

```
import datetime, xlwt
f = xlwt.Font()
f.name = 'Arial'
f.height = 240
p = xlwt.Pattern()
p.pattern = xlwt.Pattern.SOLID PATTERN
p.pattern fore colour = 0x0A
s = xlwt.XFStyle()
s.num format str = '0.00%'
s.font = f
s.pattern = p
s1 = xlwt.XFStyle()
s1.num format str = 'YYYY-MM-DD'
s1.font = f
s1.pattern = p
a = 8
b = 10
```

```
wb = xlwt.Workbook()
ws = wb.add_sheet('out1')
#以百分比的形式显示，保留两位小数
ws.write(0,3,float(a/b),s)
#显示日期
ws.row(0).write(4,datetime.date(2017,6,1),s1)
wb.save(r'D:\xunlian\out.xls')
```

将上述代码放入一个程序中，运行后，可以得到如图 5-3 所示的 out.xls 文件。

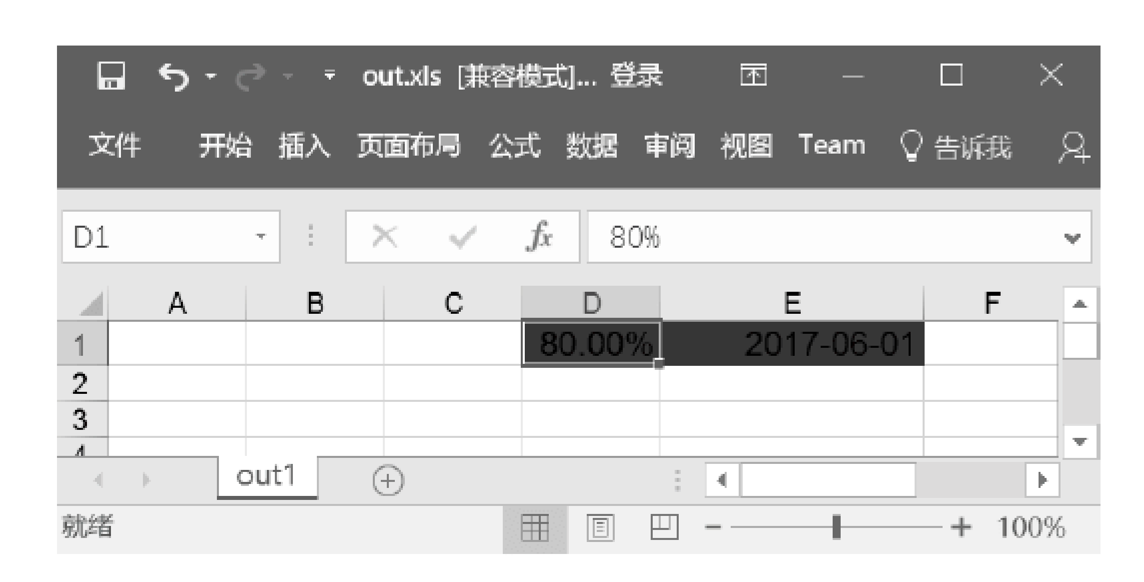


图 5-3　例 5-23 的运行结果

5.4.3　使用 xlutils 修改 Excel 文件

通过 xlrd.open_workbook()打开的 Book 对象是只读的，不能直接对其进行修改操作，而 xlwt.Workbook()返回的 Workbook 对象虽然可写，但是写的时候只能从零写起，若需要修改一个已存在数据的 Excel 文件，将如何操作呢？接下来进行介绍。

使用 xlutils.copy 中的 copy()方法，可以将一个 xlrd.Book 对象转化为一个 xlwt.Workbook 对象，这样就可以直接对已存在的 Excel 文件进行修改了。

【例 5-24】使用 xlutils 修改 Excel 文件：

```
import xlrd
import xlutils.copy
book = xlrd.open_workbook(r'D:\xunlian\out.xls',formatting_info=True)
wtbook = xlutils.copy.copy(book)
wtsheet = wtbook.get_sheet(0)
wtsheet.write(0, 0, "it has been changed.")
wtbook.save(r'D:\xunlian\out.xls')
```

将上述代码放入一个程序中，运行后，可以得到图 5-4 所示的 out.xls 文件。

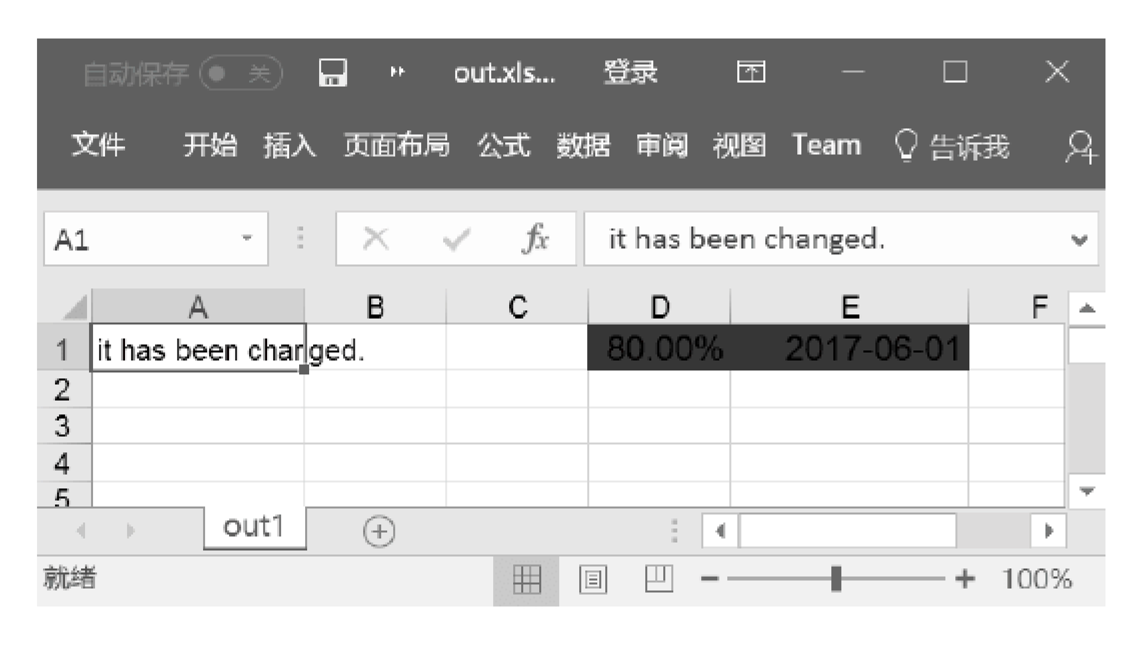


图 5-4　例 5-24 的运行结果

注意：

- 调用 xlrd.open_workbook()时，如果不指定 formatting_info=True，那么修改后整个文档的样式会丢失。对一个单元格进行 write 操作时，如果不指定样式，也会将原来的样式丢失。
- 注意调用 copy()的方法。也可以通过声明 from xlutils.copy import copy 来直接调用 copy()。

5.5 HTML 文件

HTML 是超文本标记语言 Hyper Text Markup Language 的缩写，它通过标记符来标记要显示的网页中的各个部分。网页文件本身是一种文本文件，通过在文本文件中添加标记符，可以告诉浏览器如何显示其中的内容，如怎样显示内容、如何布局等。

本节用 Beautiful Soup(Python 的一个库)来操作 HTML 文件，Beautiful Soup 最主要的功能是从网页中抓取数据。

5.5.1 Beautiful Soup 安装

在这里，我们推荐使用 Beautiful Soup 4，不过它已经被移植到 BS4 了，也就是说，我们需要使用 import bs4 命令导入 BS4，所以，这里我们用的版本是 Beautiful Soup 4.6.0 (简称 BS4)。

可以利用 pip 来安装 Beautiful Soup。其方法是：下载后缀为.whl 的 Beautiful Soup 安装包，在 DOS 命令提示符窗口下进入 Python 安装位置的 Scripts 文件夹，将安装包复制到此文件夹下，再在命令提示符窗口下运行以下命令：

```
pip install beautifulsoup4-4.6.0-py3-none-any.whl
```

接下来需要安装 lxml。其方法是：下载对应系统和版本的安装包，如后缀名为.whl，运行 pip install name.whl 即可；如后缀名为.exe，运行 easy_install name.exe 即可。

Beautiful Soup 支持 Python 标准库中的 HTML 解析器，同时也支持一些第三方的解析器，如果没有安装第三方解析器，则 Python 会使用默认的解析器，但 lxml 解析器功能更加强大，速度也更快。

5.5.2 创建 Beautiful Soup 对象

欲创建 Beautiful Soup 对象，首先需要从 bs4 中导入 BeautifulSoup 模块，使用的命令为：

```
from bs4 import BeautifulSoup
```

创建 Beautiful Soup 对象时，既可以直接创建，也可以通过本地的 HTML 文档间接创建。

1. 直接创建 Beautiful Soup 对象

使用：

```
soup = BeautifulSoup(html)
```

该语句可直接由网页字符串创建 Beautiful Soup 对象(html 是字符串，其内容为网页文本)，也可先使用 urllib 模块的 urllib.request.urlopen(url)方法打开 url 网页，再通过 BeautifulSoup(html)方法创建 Beautiful Soup 对象。例如：

```
>>> import urllib.request
>>> from bs4 import BeautifulSoup
>>> url = 'https://baike.baidu.com'
>>> html = urllib.request.urlopen(url)
>>> print(html)
<http.client.HTTPResponse object at 0x000002AA693BD278>
>>> soup = BeautifulSoup(html,"html.parser")
>>> print(soup.prettify())
<!DOCTYPE html>
<!--STATUS OK-->
<html>
 <head>
  <meta charset="utf-8"/>
  <meta content="IE=Edge" http-equiv="X-UA-Compatible">
   <meta content="always" name="referrer"/>
   <meta content="百度百科是一部内容开放、自由的网络百科全书，旨在创造一个涵盖所有
领域知识，服务所有互联网用户的中文知识性百科全书。在这里你可以参与词条编辑，分享贡
献你的知识。" name="description"/>
   <title>
    百度百科_全球最大中文百科全书
   </title>
```

其中，prettify()方法的作用是格式化 soup 对象的内容。

2. 通过本地 HTML 文档间接创建 Beautiful Soup 对象

使用本地 HTML 文档 index.html 创建 Beautiful Soup 对象的代码为：

```
a = open('index.html')
soup = BeautifulSoup(a)
```

【例 5-25】使用本地 HTML 文档创建 BeautifulSoup 对象。

有下面一段 HTML 文档(存放在 D:\xunlian 目录下，文件名为 hello.html)：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
</head>
<body>
<p class="title" name="hey"><b>it's a fantastic place</b>
<a href="http://something.com/elsie" class="sister" id="link1"><!-- Elsie --></a>
</p>
</body>
</html>
```

将上面的 HTML 文档格式化输出的代码及其运行结果如下：

```
>>> from bs4 import BeautifulSoup
>>> a = open(r'D:\xunlian\hello.html')
>>> soup = BeautifulSoup(a)
>>> print(soup.prettify())
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8"/>
  <title>
   Hello World!
  </title>
 </head>
 <body>
  <p class="title" name="hey">
   <b>
    it's a fantastic place
   </b>
   <a class="sister" href="http://something.com/elsie" id="link1">
    <!-- Elsie -->
   </a>
  </p>
 </body>
</html>
```

5.5.3 解析 HTML 文件

Beautiful Soup 将复杂的 HTML 文档转换成一个复杂的树形结构，每个节点都是 Python 对象，所有对象可以归纳为以下 4 种：

- Tag。
- NavigableString。
- BeautifulSoup。
- Comment。

1. Tag

通俗地说，Tag 就是 HTML 中的标签。例如，对于如下 title 标签而言：

```
<title>Hello World! </title>
```

Tag 指的就是首尾标志加上中间的内容。下面的例子展示了如何使用 Beautiful Soup 来获取 Tag。

【例 5-26】获取 title 标签和 head 标签：

```
>>> print(soup.title)
<title>Hello World!</title>
>>> print(soup.head)
<head>
<meta charset="utf-8"/>
<title>Hello World!</title>
</head>
```

使用“soup.标签名”可以轻松地获取指定的标签，不过找到的是符合要求的第一个标签。要查找符合要求的所有标签，可采用本节最后介绍的 find_all()方法。

Tag 有两个重要的属性：name 和 attrs，下面通过两个例子说明一下如何获取。

【例 5-27】获取 name 属性：

```
>>> print(soup.name)
[document]
>>> print(soup.head.name)
head
```

soup 对象本身比较特殊，它的 name 即为[document]，对于其他内部标签，输出的值便为标签本身的名称。

【例 5-28】获取 attrs 属性：

```
>>> print(soup.p.attrs)
{'class': ['title'], 'name': 'hey'}
```

在这里，我们把 p 标签的所有属性显示出来，得到的类型是一个字典。若想单独获取某个属性，例如获取 class，可以这样写：

```
>>> print(soup.p['class'])
['title']
```

或者使用 get 方法，传入属性名称：

```
>>> print(soup.p.get('class'))
['title']
```

2. NavigableString

前面提到获取标签内容，这里介绍一下如何获取标签内部的文字。

【例 5-29】获取标签内部文字：

```
>>> print(soup.b.string)
it's a fantastic place
```

标签内部文字类型为一个 NavigableString，即可以遍历的字符串。

3. BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容，大多数情况下，可以把它当作 Tag 对象，它是一个特殊的 Tag，我们可以分别获取它的类型、名称以及属性。

【例 5-30】获取 BeautifulSoup 对象的类型、名称和属性：

```
>>> print(type(soup.name))
<class 'str'>
>>> print(soup.name)
[document]
>>> print(soup.attrs)
{}
```

4. Comment

Comment 对象是一个特殊类型的 NavigableString 对象，其实输出的内容仍然不包括注释符号，但是如果不好好处理它，可能会给文本处理带来意想不到的麻烦。

【例 5-31】使用 comment 对象获取标签信息和属性：

```
>>> print(soup.a)
<a class="sister" href="http://something.com/elsie" id="link1"><!-- Elsie
--></a>
>>> print(soup.a.string)
 Elsie
>>> print(type(soup.a.string))
<class 'bs4.element.Comment'>
```

a 标签里的内容实际上是注释，但是，如果利用.string 来输出它的内容，我们就会发现，它已经把注释符号去掉了。

前已提及，简单的“soup.标签名”只能获取第一个符合要求的标签。本节最后介绍一下 find_all()方法，用来搜索符合条件的所有标签，并返回一个列表。

find_all()方法的语法格式为：

```
find_all(name, attrs, recursive, text, limit, **kwargs)
```

各参数的含义如下。

name：查找所有名字为 name 的 tag。此参数可以为字符串、正则表达式、列表、True 和方法。

attrs：可以用一个字典的形式指定。

recursive：默认为 True，检索当前 tag 的所有子孙节点；若指定为 False，则只检索一级子节点。

text：用于指定待搜索字符串的内容，也可支持 name 参数的几种形式，但其返回的不是对象列表而是文本列表。若 name 和 text 两个参数同时出现，则 text 会作为 name 的一个

附加条件，返回的还是带标签的列表。

limit：当 HTML 文档太大时，搜索会很慢。如果我们不需要搜索到所有的结果，可使用 limit 参数限制返回结果的数量。

**kwargs：关键字参数。

【例 5-32】使用 find_all()方法搜索所有符合条件的标签或文本：

```
>>> soup.find_all('b')                    #name为字符串形式
[<b>it's a fantastic place</b>]
>>> import re
>>> soup.find_all(re.compile('b'))        #name为正则表达式形式
[<body>
<p class="title" name="hey"><b>it's a fantastic place</b>
<a class="sister" href="http://something.com/elsie" id="link1"><!-- Elsie
--></a>
</p>
</body>, <b>it's a fantastic place</b>]
>>> soup.find_all(['a','p','b'])          #name为列表形式
[<p class="title" name="hey"><b>it's a fantastic place</b>
<a class="sister" href="http://something.com/elsie" id="link1"><!-- Elsie
--></a>
</p>, <b>it's a fantastic place</b>, <a class="sister" href="http://someth
ing.com/elsie" id="link1"><!-- Elsie --></a>]
>>> soup.find_all(attrs={'id':'link1','href':re.compile('^http')}) #attrs
参数
[<a class="sister" href="http://something.com/elsie" id="link1"><!-- Elsie
--></a>]
>>> soup.find_all(text="Hello World!")  #text参数
['Hello World!']
>>> soup.find_all(id='link1')             #关键字参数
[<a class="sister" href="http://something.com/elsie" id="link1"><!-- Elsie
--></a>]
```

5.6 XML 文件

XML 是可扩展标记语言 eXtensible Markup Language 的缩写，其中的标记是关键部分。相对于 HTML 文件来说，XML 更注重数据内容，用来传输和存储数据。

我们可以先创建内容，然后使用限定标记来标记它，从而使每个单词、短语或块成为可识别、可分类的信息。

5.6.1 解析 XML 文件

在 Python 标准库中，有三种方式来解析 XML，分别说明如下。

(1) SAX(Simple API for XML)：xml.sax 模块实现的是 SAX API，虽然速度和内存占用方面有了很大提高，但是失去了便捷性。SAX 使用事件驱动模型，在解析过程中，通过触发事件并调用用户定义的回调函数来处理 XML 文件。其特点是速度较快，占用内存比较少。

(2) DOM(Document Object Model)：将 XML 数据在内存中解析成一棵树，即 DOM 在处理之前必须把基于 XML 文件生成的树状数据置于内存，通过对树的操作来操作 XML。xml.dom 提供了几个模块，各模块性能也有所不同。其特点是速度较慢，耗内存。

(3) ElementTree(元素树)：ElementTree 就像一个轻量级的 DOM，具有方便友好的 API。其特点是代码可用性好，速度快，消耗内存少。

综合以上三种方式的特点，推荐使用 ElementTree 方式解析 XML。ElementTree 提供了两个对象将 XML 文档解析成树：ElementTree 将整个 XML 文档转化为树，Element 则代表树上的单个节点。对整个 XML 文档的交互(读取、写入、查找)一般是在 ElementTree

层面进行的；对单个 XML 元素及其子元素，则是在 Element 层面进行的。

下面介绍如何利用 ElementTree 解析 XML。

【例 5-33】本节用到的 XML 文件内容如下：

```
D:\xunlian\myxml.xml
<?xml version="1.0"?>
- <doc>
    <branch hash="2g67ed90" name="codingpy.com"> text01,source </branch>
  - <branch hash="h34900em" name="free00">
        <sub-branch name="subfree00"> xml,sgml </sub-branch>
    </branch>
    <branch name="invalid"> </branch>
  </doc>
```

将下述代码放入一个程序中，以便运行：

```
#首先，加载文档
import xml.etree.ElementTree as ET
t = ET.ElementTree(file=r'D:\xunlian\myxml.xml')
#获取根元素(root element)
t.getroot()
#查看根元素的属性
root=t.getroot()
print(root.tag, root.attrib)
#遍历根元素的子元素
for child of root in root:
    print(child of root.tag, child of root.attrib)
#使用索引访问特定子元素
print(root[0].tag, root[0].text)
#查找 XML 文档中所有元素，利用 Element 对象中 iter 方法实现
for elem in t.iter():
    print(elem.tag, elem.attrib)
#使用 iter 方法任意遍历某一 tag
for elem in t.iter(tag='branch'):
    print(elem.tag,elem.attrib)
#使用 XPath 查找元素，Element 对象中有一些 find 方法接受 XPath 路径或某一属性为参数
for elem in t.iterfind('branch/sub-branch'):
    print(elem.tag,elem.attrib)
for elem in t.iterfind("branch[@name='free00']"):
    print(elem.tag,elem.attrib)
```

运行结果如下：

```
doc {}
branch {'name': 'codingpy.com', 'hash': '2g67ed90'}
branch {'name': 'free00', 'hash': 'h34900em'}
branch {'name': 'invalid'}
branch
text01,source

doc {}
branch {'name': 'codingpy.com', 'hash': '2g67ed90'}
branch {'name': 'free00', 'hash': 'h34900em'}
sub-branch {'name': 'subfree00'}
branch {'name': 'invalid'}
branch {'name': 'codingpy.com', 'hash': '2g67ed90'}
branch {'name': 'free00', 'hash': 'h34900em'}
branch {'name': 'invalid'}
sub-branch {'name': 'subfree00'}
sub-branch {'name': 'subfree00'}
```

需要注意的是，iterfind 方法会返回一个匹配所有元素的迭代器。ElementTree 中还有

find 方法和 findall 方法，find 方法会返回第一个与 XPath(XPath 是一门在 XML 文档中查找信息的语言，使用路径表达式在 XML 文档中选取节点)匹配的子元素，findall 方法以列表形式返回所有匹配的子元素。

5.6.2 创建 XML 文件

本节介绍如何利用 ElementTree(ET)完成 XML 文档的构建。ElementTree 对象的 write 方法就可以实现这个需求。

一般来说，有两种主要使用场景。一是先读取一个 XML 文档进行修改，然后再将修改写入文档，二是从头创建一个新 XML 文档。

【例 5-34】通过 Element 来修改上例中的 XML 文档：

```
import xml.etree.ElementTree as ET
t = ET.ElementTree(file=r'D:\xunlian\myxml.xml')
root = t.getroot()
del root[2]
root[0].set('foo','bar')
for subelem in root:
    print(subelem.tag,subelem.attrib)
t.write(r'D:\xunlian\myxml.xml')
```

将上述代码放入一个程序中，运行结果为：

```
branch {'name': 'codingpy.com', 'hash': '2g67ed90', 'foo': 'bar'}
branch {'name': 'free00', 'hash': 'h34900em'}
>>>
```

在上面的代码中，我们删除了 root 元素的第三个子元素，为第一个子元素增加了新属性。这个树可以重新写入文件中。最终的 XML 文档如图 5-5 所示。

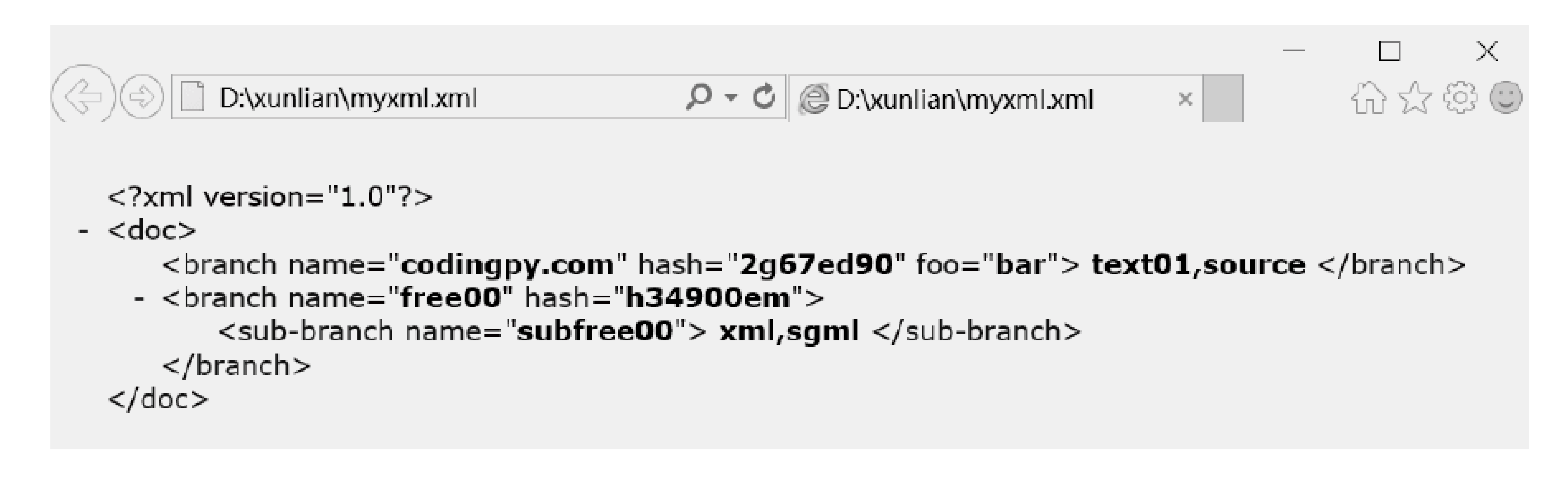


图 5-5　例 5-34 的运行结果

如果是从头构建一个完整的文档，ElementTree 模块提供了一个 SubElement 工厂函数，使创建元素的过程变得很简单。

【例 5-35】使用 SubElement 函数构建 XML 文档：

```
import xml.etree.ElementTree as ET
a = ET.Element('elem')
c = ET.SubElement(a,'child1')
c.text = 'sometext'
d = ET.SubElement(a,'child2')
b = ET.SubElement(d,'elem_b')
```

```
root = ET.Element('root')
root.extend((a,b))
tree = ET.ElementTree(root)
tree.write(r'D:\xunlian\first.xml')
```

将上述代码放入一个程序中，生成的 XML 文档如图 5-6 所示。

```
D:\xunlian\first.xml    D:\xunlian\first.xml

  <?xml version="1.0"?>
- <root>
  - <elem>
      <child1>sometext</child1>
    - <child2>
        <elem_b/>
      </child2>
    </elem>
    <elem_b/>
  </root>
```

图 5-6　例 5-35 的运行结果

5.7　异 常 处 理

5.7.1　异常

异常是一个事件，此事件会在程序执行过程中发生，影响程序的正常执行。一般情况下，Python 在无法正常处理程序时就会产生异常。

Python 用异常对象(exceptionobject)表示异常情况。当发生异常时，我们需要捕捉它，否则程序会用回溯(traceback)的方式停止运行。

在 Python 中，标准异常情况如表 5-4 所示。

表 5-4　异常情况

名　称	说　明
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零(所有数据类型)

续表

名　称	说　明
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入，到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于 Python 解释器不是致命的)
NameError	未声明/初始化对象(没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weakreference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtimebehavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

5.7.2　try、else、finally 语句

捕捉异常可以使用 try、except、else、finally 语句。

try/except 语句用来检测 try 语句块中的错误，从而使 except 语句捕捉异常信息并处理，若使程序不会在异常发生时就停止运行，只需在 try 中捕捉它。

try/except/else 的语法如下：

```
try:
   <语句>                    #运行别的代码
except <name:>
   <语句>                    #如果在 try 部分引发了 name 异常
except <name>,<数据:>
   <语句>                    #如果引发了 name 异常，获得附加的数据
else:
   <语句>                    #如果没有异常发生
finally:
   <语句>                    #不管 try 子句内部是否有异常发生，都会执行语句
```

try 语句的工作流程如下。

(1) 当遇到一个 try 语句后，Python 就在当前程序的上下文做标记，当出现异常时可以较快地回到这里，再执行 try 子句，然后执行什么取决于运行过程中是否出现异常。

(2) 如果当 try 后的语句执行时发生异常，Python 就跳回到 try 并执行第一个匹配该异常的 except 子句。异常处理完毕后就继续运行(除非在处理异常时又引发新的异常)。

(3) 如果在 try 后的语句里发生了异常，却没有匹配的 except 子句，异常将被提交到上层的 try，或者到程序的最上层(这样将结束程序，并显示默认的出错信息)。

(4) 如果在 try 子句执行时没有异常发生，Python 将执行 else 语句后的语句(如果有 else 的话)，然后程序通过整个 try 语句并继续运行。

(5) 不论是否发生异常，finally 子句一定会被执行。

【例 5-36】使用 try/except 关键字捕捉异常：

```
try:
   print(8/0)
except ZeroDivisionError:
   print('除数不能为0')
```

将上述代码放入一个程序中，运行结果为：

```
除数不能为0
>>> |
```

一个 except 语句只能捕捉其后声明的异常类型，但如果抛出的是其他类型的异常，就需要再增加一个 except 语句了。当然，也可以指定一个更加通用的异常类型，比如 Exception。除了声明多个 except 语句外，也可以在一个 except 语句中将多个异常作为元组列出来。

【例 5-37】捕捉多个异常，并将多个异常以元组形式列出：

```
try:
   print(8/'0')
```

```
except(ZeroDivisionError,Exception):
    print('发生了一个异常')
```

将上述代码放入一个程序中，运行结果为：

```
发生了一个异常
>>> |
```

使用 finally 子句时，不管 try 子句内部是否有异常，都会执行 finally 子句，所以 finally 子句用于关闭文件或网络套接字(第 12 章介绍)时会非常有用。在同一条语句中可以组合使用 try、except、finally 和 else。

【例 5-38】组合使用 try/except/else/finally 子句进行异常处理：

```
try:
    print(8/'0')
except(ZeroDivisionError,Exception):
    print('发生了一个异常')
else:
    print('正常运行')
finally:
    print('cleaning up')
```

将上述代码放入一个程序中，运行结果为：

```
发生了一个异常
cleaning up
>>> |
```

5.7.3 触发异常和自定义异常

异常可以在某些地方出错时自动引发，下面介绍一下如何自己引发异常，并且介绍一下如何创建自己的异常类型。

在 Python 中使用 raise 关键字触发异常：

```
def ThorwErr():
    raise Exception("抛出一个异常")
#Exception:抛出一个异常
ThorwErr()
```

raise 关键字触发的是一个通用的异常类型(Exception)，一般来说，触发的异常越详细越好，Python 中内建了很多异常类型，可以通过 dir 函数查看异常类型。

【例 5-39】使用 dir 函数查看 Python 内建模块 builtins 中的异常类型：

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', '
BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'Child
ProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRef
usedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ell
ipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'File
NotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'I
OError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessL
ookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'Runti
meError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'Syntax
Error', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Timeout
Error', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'U
nicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarni
ng', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivision
Error', '  build class  ', '  debug  ', '  doc  ', '  import  ', '  loader
```

这里我们只截取了异常类部分，内建模块 builtins 还包括很多其他类型和方法。

虽然内建的异常类已经包含大部分情况，可以满足很多要求，但有时候还是需要创建自己的异常类(类的概念在第 6 章中介绍)。

在 Python 中，可以自定义特殊类型的异常，但前提是确保从 Exception 类继承(不管是直接继承还是间接继承)。

【例 5-40】自定义异常类：

```
class SomeCustomException(Exception):
    pass
```

5.7.4　使用 sys 模块返回异常

在 Python 中，另一种获取异常信息的方式是通过 sys 模块中的 exc_info()函数，此函数会返回一个三元组：(异常类，异常类的实例，跟踪记录对象)。

【例 5-41】使用 exc_info()函数返回异常：

```
try:
    8/0
except:
    import sys
    t = sys.exc info()
    print(t)
    for i in t:
        print(i)
```

将上述代码放入一个程序中，运行结果为：

```
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',), <tra
ceback object at 0x0000023FFC730308>)
<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x0000023FFC730308>
>>> |
```

5.8　使用 pdb 模块调试程序

在 Python 中，语法错误可以由 Python 解释器发现，但逻辑错误或变量使用错误却不容易发现，若结果不符合预期，则需要进行调试。1.2.3 小节曾介绍了使用 IDLE 自身的功能调试程序的方法。其实，Python 自带的 pdb 模块也是一个很好的调试工具，使用它可以为脚本设置断点、单步执行、查看变量值等。

pdb 可以使用 import 导入，也可以通过命令行参数方式启动。

【例 5-42】使用 dir 函数查看 pdb 模块的内建函数：

```
>>> import pdb
>>> dir(pdb)
['Pdb', 'Restart', 'TESTCMD', '__all__', '__builtins__', '__cached__', '__
doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'_rstr', '_usage', 'bdb', 'cmd', 'code', 'dis', 'find_function', 'getsource
lines', 'glob', 'help', 'inspect', 'lasti2lineno', 'line_prefix', 'linecac
he', 'main', 'os', 'pm', 'post_mortem', 'pprint', 're', 'run', 'runcall',
'runctx', 'runeval', 'set_trace', 'signal', 'sys', 'test', 'traceback']
>>> |
```

5.8.1 常用的 pdb 函数

1. pdb.run()函数

pdb.run()函数主要用于调试语句块，其基本语法如下：

```
run(statement,globals=None, locals=None)
```

参数含义：statement - 要调试的语句块，以字符串形式表示。

globals - 可选参数，设置 statement 运行的全局环境变量。

locals - 可选参数，设置 statement 运行的局部环境变量。

【例 5-43】使用 pdb.run()函数调试语句块：

```
import pdb                        #导入调试模块
pdb.run('''                       #调用run()函数执行一个for循环
for t in range(0,10,3):
    t += 2
    print(t)
''')
> <string>(2)<module>()
(Pdb) n                           #(Pdb)为调试命令提示符，表示可输入调试命令
> <string>(3)<module>()
(Pdb) n                           #n表示执行下一行
> <string>(4)<module>()
(Pdb) n                           #程序循环一次后，输出一个结果
2
> <string>(2)<module>()
(Pdb) continue                    #continue表示继续执行程序
5
8
11
```

2. pdb.runeval()函数

pdb.runeval()函数主要用于调试表达式，其基本语法如下：

```
runeval(expression,globals=None,locals=None)
```

参数含义：expression - 要调试的表达式。

globals - 可选参数，设置 expression 运行的全局环境变量。

locals - 可选参数，设置 expression 运行的局部环境变量。

【例 5-44】使用 pdb.runeval()函数调试表达式：

```
import pdb
pdb.runeval('(3+5)*2 -6')         #使用runeval()函数调试表达式'(3+5)*2 -6'
> <string>(1)<module>()->2
(Pdb) n                           #使用n命令单步执行
--Return--
> <string>(1)<module>()->10
(Pdb) n                           #得出表达式的值
10
```

3. pdb.runcall()函数

pdb.runcall()函数主要用于调试，其基本语法如下：

```
runcall(*args,**kwds)
```

参数含义：args(kwds) - 函数参数。

【例 5-45】使用 pdb.runcall()函数调试：

```
import pdb
def sum(*args):                      #定义函数sum，求所有参数之和
    total = 0
    for value in args:
        total += value
    return total
pdb.runcall(sum, 2,4,6,8,10)         #使用runcall()调试函数
> <stdin>(2)sum()
(Pdb) n                              #进入调试状态，单步执行
> <stdin>(3)sum()
(Pdb) n
> <stdin>(4)sum()
(Pdb) n
> <stdin>(3)sum()
(Pdb) n
> <stdin>(4)sum()
(Pdb) n
> <stdin>(3)sum()
(Pdb) n
> <stdin>(4)sum()
(Pdb) n
> <stdin>(3)sum()
(Pdb) n
> <stdin>(4)sum()
(Pdb) n
> <stdin>(3)sum()
(Pdb) n
> <stdin>(4)sum()
(Pdb) n
> <stdin>(3)sum()
(Pdb) n
> <stdin>(5)sum()
(Pdb) n
--Return--
> <stdin>(5)sum()->30
(Pdb) continue                         #继续执行
30                                     #函数最后返回结果
```

4. pdb.set_trace()函数

pdb.set_trace()函数主要用于在脚本中设置硬断点，其基本语法如下：

```
set_trace()
```

【例 5-46】使用 pdb.set_trace()函数设置硬断点：

```
import pdb

pdb.set_trace()                         #设置硬断点
for i in range(6):
    i *= i
    print(i)
> d:\project\pdb_04.py(4)<module>()
-> for i in range(6):
(Pdb) list                              #使用list列出脚本内容
  1     import pdb
  2
  3     pdb.set_trace()
  4  -> for i in range(6):
  5         i *= i
  6         print(i)
[EOF]                                   #列出脚本内容结束标志
(Pdb) continue                          #继续执行，输出最后结果
0
1
4
9
16
25
```

5.8.2 pdb 调试命令

pdb 模块中的调试命令可以完成单步执行、打印变量值、设置断点等功能，主要命令如表 5-5 所示。

表 5-5 pdb 模块中的调试命令

#完整命令	简写命令	描 述
#args	a	打印当前函数的参数
#break	b	设置断点
#clear	cl	清除断点
#condition	无	设置条件断点
#continue	c	继续运行，直到遇到断点或者脚本结束
#disable	无	禁用断点
#enable	无	启用断点
#help	h	查看 pdb 帮助
#ignore	无	忽略断点
#jump	j	跳转到指定行运行
#list	l	列出脚本清单
#next	n	执行下条语句，遇到函数不进入其内部
#print	p	打印变量值
#quit	q	退出 pdb
#return	r	一直运行到函数返回
#tbreak	无	设置临时断点，断点只中断一次
#step	s	执行下一条语句，遇到函数进入其内部
#where	w	查看所在的位置
#!	无	在 pdb 中执行语句

【例 5-47】使用 pdb 调试命令调试程序。

首先我们编写一个求解水仙花数的程序，然后调出调试命令行，使用表 5-5 所示的调试命令进行调试。

```
def narci_num(a,b):                     #定义水仙花数函数
    for i in range(a,b):
        h = i//100
        d = i//10%10
        u = i%10
        if i == h**3 + d**3 + u**3:
            print(i)

narci_num(100,1000)                     #计算100-999之间的水仙花数

d:\Project>python -m pdb pdb_05.py      #运行此命令进行程序调试
> d:\Project\pdb_05.py(1)<module>()
-> def narci_num(a,b):
(Pdb) list                              #list默认只列出11行
```

```
  1  -> def narci_num(a,b):
  2         for i in range(a,b):
  3             h = i//100
  4             d = i//10%10
  5             u = i%10
  6             if i == h**3 + d**3 + u**3:
  7                 print(i)
  8
  9      narci_num(100,1000)
 10
 11                                         #11行后如有代码，可以使用'l page1,page2'列出
(Pdb) b 2                                   #使用break命令设置第2行为断点
Breakpoint 1 at d:\Project\pdb_05.py:2      #返回断点编号1
(Pdb) b 7
Breakpoint 2 at d:\Project\pdb_05.py:7
(Pdb) c
> d:\Project\pdb_05.py(2)narci_num()
-> for i in range(a,b):
(Pdb) c                                     #使用c命令运行脚本
> d:\Project\pdb_05.py(2)narci_num()
-> for i in range(a,b):
(Pdb) n                                     #单步执行
> d:\Project\pdb_05.py(3)narci_num()
-> h = i//100
(Pdb) n
> d:\Project\pdb_05.py(4)narci_num()
-> d = i//10%10
(Pdb) n
> d:\Project\pdb_05.py(5)narci_num()
-> u = i%10
(Pdb) n
> d:\Project\pdb_05.py(6)narci_num()
-> if i == h**3 + d**3 + u**3:
(Pdb) n
> d:\Project\pdb_05.py(2)narci_num()
-> for i in range(a,b):
(Pdb) p i                                   #打印i的值
101
(Pdb) disable 2                             #禁用断点2
Disabled breakpoint 2 at d:\Project\pdb_05.py:7
(Pdb) c
> d:\Project\pdb_05.py(2)narci_num()
-> for i in range(a,b):
(Pdb) enable 2                              #恢复断点2
Enabled breakpoint 2 at d:\Project\pdb_05.py:7
(Pdb) c
> d:\Project\pdb_05.py(2)narci_num()
-> for i in range(a,b):
(Pdb) cl                                    #清除所有断点，输入y确认
Clear all breaks? y
Deleted breakpoint 1 at d:\Project\pdb_05.py:2
Deleted breakpoint 2 at d:\Project\pdb_05.py:7
(Pdb) c                                     #继续运行
153
370
371
407
The program finished and will be restarted
> d:\Project\pdb_05.py(1)<module>()
-> def narci_num(a,b):
(Pdb) q                                     #使用q命令退出pdb调试
```

5.9　案例实训：文本文件的操作与异常处理

综合前面所学内容，本章最后给出一个综合性实例。本例结合了文本文件的操作与异常事件的处理，使用 re.sub()函数对文本文件中的字符串进行替换，并将替换后的内容存入另一个文本文件。本例的代码如下：

```
import re
try:
    infile = input("请输入文本文件名(含完整路径)：")
```

```
    f1 = open(infile,'r')
    outfile = input("请输入另存为的文件名(含完整路径): ")
    f2 = open(outfile,'w')

    sourcestr = input("请输入要替换的原文本内容: ")
    targetstr = input("请输入替换后的文本内容: ")
    n = 0
    p = 0

    #读取源文件内容，逐行替换文本内容，写入新文件，并统计替换次数。
    for line in f1.readlines():
        p += 1
        if sourcestr in line:                    #判断 sourcestr 是否在文本行中
            replace = re.sub(sourcestr,targetstr,line)
              #将 sourcestr 替换为 targetstr，写入行数据

            n += line.count(sourcestr)        #统计替换次数
            f2.write(replace)            #将替换后的文本逐行写入新文件中
            continue
        elif n != 0:
            f2.write(line)
            print("第%s 行中没有要替换的内容。"%p)
        else:
            print("要替换的内容不存在。")

    print("替换的次数: %s"%n)
except FileNotFoundError:                        #捕捉文件不存在的异常
    print("File not found.")
except PermissionError:                         #捕捉文件权限的异常
    print("You don't have permission to access this file.")
finally:
    f1.close()
    f2.close()
    print("End up.")
```

在 try 子句中，先输入替换前后文本文件的完整路径及文件名，再输入需要替换的文本以及替换后的文本。通过 for 循环逐行读取源文件内容，在循环内首先用 if 语句判断该行文本中是否包含 sourcestr，若包含，则继续执行；使用 sub()函数替换文本，统计替换次数，并向新文件中写入替换后的文本。若文件中存在 sourcestr，但某行中不存在，则执行 elif 分支，将当前行原文本写入新文件，并输出提示信息。当文件中不存在 sourcestr 时，执行 else 分支，输出提示信息“要替换的内容不存在”。当输入的文件路径或文件名不存在时，except 语句捕捉 FileNotFoundError 异常，并输出提示信息。因文件权限问题造成无法读取或无法写入文件时，except 语句捕捉 PermissionError 异常，输出提示信息。无论是否出现异常，最终都会执行 finally 子句并输出“End up.”。

此案例中，我们将对一段描写抗日战争史的文字进行替换，将“八年抗战”替换为“十四年抗战”，此文件保存在 D:\test 目录下，文本内容如下：

```
test.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
抗日战争是1931年9月至1945年8月，中国人民进行的八年反抗日本帝国主义侵
略的伟大的民族革命战争，也是一百多年来中国人民反对外敌入侵第一次取得
完全胜利的民族解放战争。这场战争是以国共两党合作为基础，有社会各界、
各族人民、各民主党派、抗日团体、社会各阶层爱国人士和海外侨胞广泛参加
的全民族抗战。中国的抗日战争是第二次世界大战的重要组成部分。
从敌我力量发展变化的趋势来看。中国八年抗战及其胜利，是百多年来帝国主
义在中国统治走向灭亡的重要转折点。成为中华民族由衰败到重新振起的伟大
转折。
```

第一种情况，替换“八年抗战”为“十四年抗战”，并将替换后的文本保存到同一目录下的 test0.txt 文件中。运行结果如下：

```
请输入文本文件名（含完整路径）：D:\test\test.txt
请输入另存为的文件名（含完整路径）：D:\test\test0.txt
请输入要替换的原文本内容：八年
请输入替换后的文本内容：十四年
替换的次数：2
End up.
```

因为原文中有两处“八年”，故替换的次数为 2，替换成功。

第二种情况，替换“基础”为“基石”，因原文第二行中没有替换内容，所以执行一次 elif 分支，运行结果如下：

```
请输入文本文件名（含完整路径）：D:\test\test.txt
请输入另存为的文件名（含完整路径）：D:\test\test1.txt
请输入要替换的原文本内容：基础
请输入替换后的文本内容：基石
第2行中没有要替换的内容。
替换的次数：1
End up.
```

第三种情况，要替换的字符串不在源文件中，没有内容替换，所以新文件内容为空。运行结果如下：

```
请输入文本文件名（含完整路径）：D:\test\test.txt
请输入另存为的文件名（含完整路径）：D:\test\test2.txt
请输入要替换的原文本内容：@
请输入替换后的文本内容：!
要替换的内容不存在。
要替换的内容不存在。
替换的次数：0
End up.
```

因为原文中有两个自然段，每个自然段以换行符结束，故相当于两“行”，所以在循环过程中，else 语句会执行两次，输出两次“要替换的内容不存在”。

第四种情况，输入的源文件不存在，则引发 FileNotFoundError 异常，输出“File not found.”。运行结果如下：

```
请输入文本文件名（含完整路径）：D:\test\hello.txt
File not found.
End up.
```

第五种情况，因文件权限问题引发 PermissionError 异常，因运行此例的电脑 C 盘文件有写权限，所以将新文件写在 C 盘时会引发异常。运行结果如下：

```
请输入文本文件名（含完整路径）：D:\test\test.txt
请输入另存为的文件名（含完整路径）：C:\test.txt
You don't have permission to access this file.
End up.
```

本 章 小 结

本章首先介绍了通过文件对象和 os、shutil 模块中的方法对文件和目录进行读写等操作；其次介绍了对 4 种特殊文件的操作，即——使用 CSV 包对 CSV 文件进行读写操作，使用 xlrd、xlwt 和 xlutils 库读写与修改 Excel 文件，使用 Beautiful Soup 库解析 HTML 文档，以三种方式解析 XML 文档以及利用 ET 构建 XML 文档；然后介绍了 Python 的标准异常对象，捕捉异常的 try、else、finally 语句，使用 raise 关键字触发异常，用户自定义异常类等；最后介绍了使用 pdb 模块调试程序，以及常用的 pdb 函数和调试命令。

习　　题

1. 填空题

(1) 获取文件指针的方法是(　　　)，当偏移相对位置为(　　　)时，offset 必须为 0 或正数。

(2) os 模块和 shutil 模块都有删除目录的方法，其中，(　　　)只能删除空目录，而(　　　)对空目录和非空目录均可删除。

(3) 使用 Beautiful Soup 解析 HTML 文档的时候，解析后的节点对象有四种，它们分别为(　　　)、(　　　)、(　　　)、(　　　)。

2. 选择题

(1) 在高级语言中，对文件操作的一般步骤是(　　)。

A. 操作-修改-关闭　　B. 读写-打开-关闭

C. 打开-操作-关闭　　D. 读-写-关闭

(2) 下列程序的输出结果是(　　):

```
import os
f = open(r'D:\test.txt','w+')
f.write('Python')
f.seek(0)
m = f.read(3)
print(m)
f.close()
```

A. Pyt　　B. Python　　C. yth　　D. Py

(3) 对 Excel 操作的说法，错误的是(　　)。

A. xlrd 库提供读取 Excel 文件的方法

B. xlwt 库提供写 Excel 文件的方法

C. xlutils 库用于修改 Excel 文件

D. 写 Excel 文件可以用 writer 方法

(4) 以下程序的正确输出是(　　)。

```
try:
    x=4/10
except ZeroDivisionError:
    print('4')
```

A. 0　　B. 4　　C. 0.4　　D. 空

3. 问答题

(1) 简单解释一下文本文件和二进制文件的区别。
(2) 简单分析本章介绍的三种解析 XML 方式的区别。
(3) 异常和错误有何区别?
(4) try-except 和 try-finally 有何区别?
(5) 使用 pdb 模块进行 Python 程序调试主要有哪几种用法?

4. 实践操作题

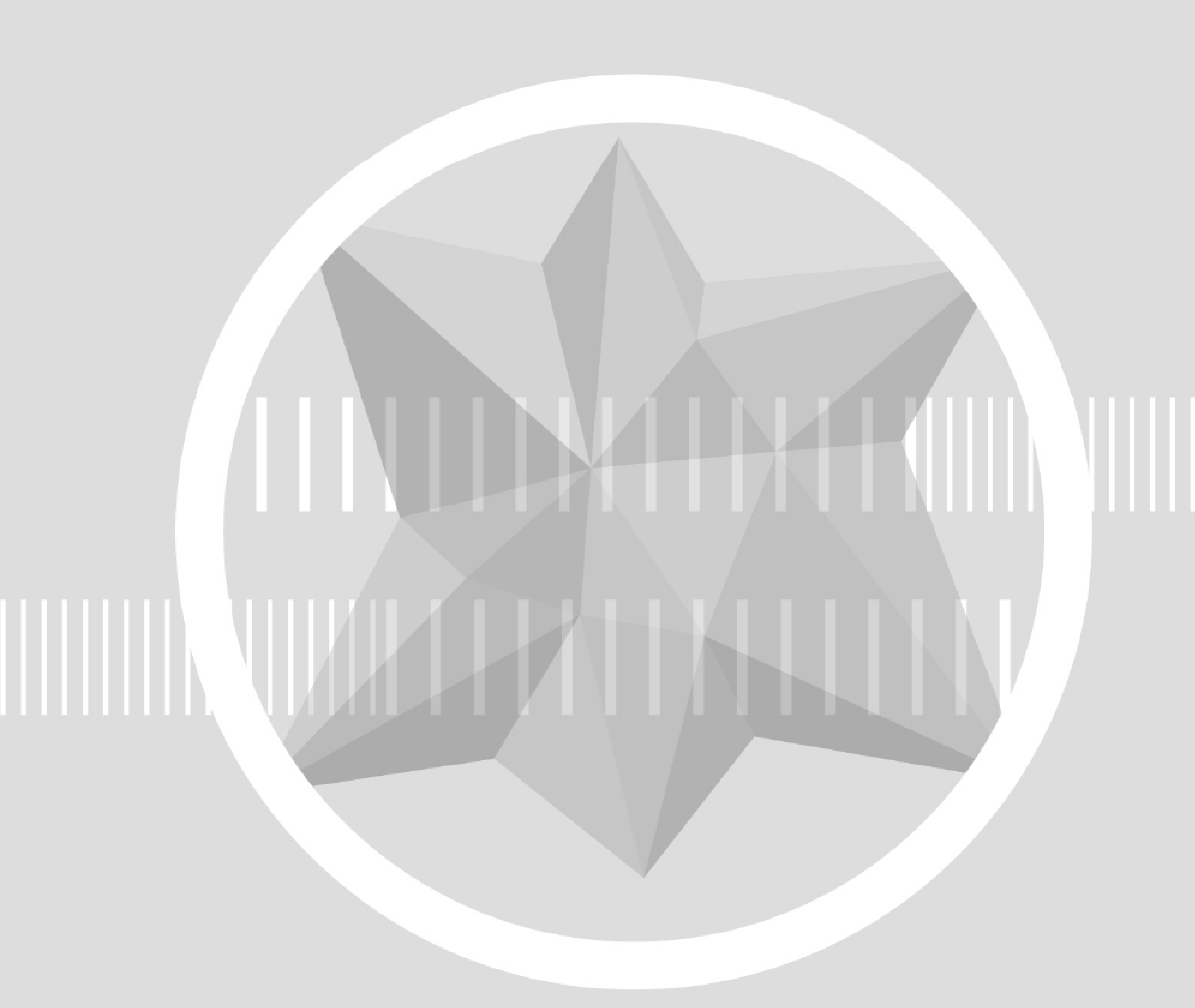
(1) 假设在 D 盘下有一个名为 some.txt 的文件。
① 写入文本“This is my first Python program”。
② 读取文本的 5 个字节。
③ 将文本文件移动到 C 盘下，并验证是否移动成功(当前盘为 C)。
(2) 获取当前工作空间目录，并获取当前目录下的所有内容。
(3) 在 Excel 中创建 csv 文件，命名为 sec.csv。
① 编程将下面三行数据写入文件 sec.csv。

a　1　2　3
b　4　5　6
c　7　8　9

② 读取该文件内容。

(4) 将输入的字符串写入 text.txt 文件中，直到输入 E 结束，如果输入 Ctrl+Z，则终止程序运行。注意要保证打开的文件能正常关闭。

(5) 将 math.sqrt()进行改进。虽然 math 模块包含大量的用于处理数值运算的函数和常量，但是，它不能识别复数，所以创建了 cmath 模块来支持复数相关运算。请创建一个 safe_sqrt()函数，它封装 math.sqrt()并能对负数进行开方运算(返回一个对应的复数)。



第 6 章

面向对象编程

本章要点

(1) 面向对象技术简介。
(2) 类与对象的定义和使用。
(3) 类的属性与方法。
(4) 类的作用域与命名空间。
(5) 类的单继承和多继承。
(6) 特定函数介绍。
(7) 面向对象程序设计应用举例。

学习目标

(1) 通过 OOP 进一步掌握模块化编程的程序设计风格。
(2) 理解在编写代码的过程中如何隐藏程序实现的细节。
(3) 学会将数据与其上的操作分离。
(4) 理解抽象程序设计风格。
(5) 掌握运用面向对象技术解决实际问题的方法。

本章我们将详细介绍 Python 的面向对象编程(Object-Oriented Programming，OOP)。

Python 从设计之初就已经是一门面向对象的语言了，正因为如此，在 Python 中创建一个类和对象是很容易的。随着软件项目规模的扩大和复杂度的增加，参与项目的开发人员越来越多，同时，整个程序的关联性和依赖性呈指数级增加。一个程序员在代码某处所做的微小改变，可能会使整个项目的开发都因此而做出较大的调整，解决这些问题的重要方法就是面向对象技术。

如果读者以前没有接触过面向对象的编程语言，那可能需要先了解面向对象语言的一些基本特征，在头脑里形成一个基本的面向对象的概念，这样有助于更容易地学习 Python 面向对象编程。接下来，我们先来简单地了解一下面向对象的一些基本概念和特征。

- 类(class)：用来描述具有相同属性和方法的对象的集合，它定义该集合中每个对象所共有的属性和方法，对象是类的实例。
- 类变量：类变量在整个实例化的对象中是公用的，类变量定义在类中且在函数体之外，类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量用于处理类及其实例对象的相关数据。
- 方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫作方法的重写(override)，也称为方法的覆盖。
- 实例变量：定义在方法中的变量，只作用于当前实例的类。
- 继承：即一个派生类(derived class)继承基类(base class)的数据成员和方法，继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 Dog 类型的对象派生自 Animal 类，这是模拟“是一个(is-a)”关系(Dog 是一种 Animal)。
- 实例化：创建一个类的实例，即类的具体对象。
- 方法：类中定义的函数。

- 对象：对象包括两类数据成员(类变量和实例变量)和方法，通过类定义的数据结构进行实例化。

与其他编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。Python 中的类提供面向对象编程的所有基本功能：类的继承机制允许继承多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法，对象可以包含任意数量和类型的数据。与其他语言不同的是，在 Python 中，一切皆为对象，只是类型有所不同，例如，“hello Python!”是一个字符串(str)类型的对象，[1,2,3]是一个列表(list)类型的对象，而 12 是一个整型(int)的对象。

6.1　类的定义与使用

Python 中的类是一个抽象的概念，甚至比函数还要抽象。我们可以把它简单地看作是数据以及由存取、操作这些数据的方法所组成的一个集合。类是 Python 的核心概念，是面向对象编程的基础。在前面的章节里我们学习了函数的用法，讲解了如何重用代码，那为什么还要用类来取代函数呢？

因为类有如下优点。

- 类对象是多态的：也就是具有多种形态，这意味着我们可以对不同的类对象使用同样的操作方法，而不需要额外编写代码。
- 类的封装：类封装之后，可以直接调用类的对象来操作内部的一些类方法，不需要让使用者看到代码工作的细节。
- 类的继承：类可以从其他类或者基类中继承它们的方法，直接使用。

6.1.1　类的定义

类是对现实世界中一些事物的封装，所有类的开头都要包括关键字 class，紧接着的是类名和冒号，随后是定义类的类体代码。类定义的语法格式如下：

```
class ClassName:
    """documentation string"""
    <statement 1>
    <statement 2>
    ...
<statement-N>
```

object 是“所有类之父”。如果你的类没有继承任何其他父类，object 将作为默认的父类，它位于所有类继承结构的最上层(继承的概念在本章稍后介绍)。定义一个类可以采用下面的方式。

【例 6-1】类的定义：

```
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性，私有属性在类外部无法直接进行访问
```

```
    weight = 0
   #定义构造方法
   def  init  (self,n,a,w):
      self.name = n
      self.age = a
      self.  weight = w
   #定义类本身的方法
   def speak(self):
      print("%s is speaking: I am %d years old." %(self.name,self.age))
#类调用
p = people('Tom',10,30)
p.speak()
```

执行以上代码，输出结果为：

```
Tom is speaking: I am 10 years old.
>>>
```

注意： 在上面的例子中，name 和 age 是类的公有属性。__weight 使用两个下划线开头，表示该属性被声明为私有属性，它不能在类的外部被使用或直接访问，但可以在类内部使用 self.__weight 来调用。

【例 6-2】(接上例)访问类的属性：

```
print(p.name)
print(p.__weight)
```

执行以上代码，输出结果为：

```
Tom
Traceback (most recent call last):
  File "D:\Python34\examples.py", line 20, in <module>
    print(p.__weight)
AttributeError: 'people' object has no attribute '__weight'
>>>
```

出错的原因在于，在类的外部使用了私有属性。

6.1.2　类属性与方法

1. 类的公有属性

public_attrs：符合正常的变量命名规则，开头没有下划线，在类的外部可以直接进行访问，如上例中的 name、age。

2. 类的私有属性

__private_attrs：由两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。在类内部的方法中使用时的格式为 self.__private_attrs。

【例 6-3】访问类的私有属性：

```
class Counter:
     privateCount = 1  #私有属性
   publicCount = 1    #公有属性
```

```
    def count(self):
        self.  privateCount += 1
        self.publicCount += 1
        print (self.  privateCount)
counter 1 = Counter()
counter 1.count()              #打印数据
print (counter 1.publicCount)     #打印数据
print (counter_1.__privateCount)  #报错，实例不能访问私有属性
```

执行以上代码，输出结果为：

```
2
2
Traceback (most recent call last):
  File "D:/Python34/aaa程序/ex6-3.py", line 12, in <module>
    print (counter_1.__privateCount)  # 报错，实例不能访问私有属性
AttributeError: 'Counter' object has no attribute '__privateCount'
>>>
```

出错的原因与上例一样。

3. 类的构造方法

__init__()：叫作构造函数或者构造方法，它在生成一个对象时被自动调用。在例 6-1 中，p=people('Tom',10,30)语句就是调用__init__()方法，将参数传递给 self.name、self.age 和 self.__weight。

4. 类的公共方法

public_method()：在类的内部，使用 def 关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数 self，且为第一个参数。self 在 Python 里不是关键字，它代表当前对象的地址，类似于 Java 语言中的 this。另外，self 不一定要写成 self，例如将 self 写成 this 在例 6-1 中也可以正确运行，但出于 Python 的代码规范，不建议用其他标识符替换 self，以免造成理解错误。

5. 类的私有方法

__private_method()：由两个下划线开头，声明该方法为私有方法，不能在类的外部调用。在类的内部调用时格式为 self.__private_methods()。

【例 6-4】类的私有方法：

```
class Site:
def   init  (self, name, url):
    self.name = name    #公有属性
    self.  url = url    #私有属性
def printme(self):
    print('name : ', self.name)
    print('url : ', self.  url)
def   printme 1(self):          #私有方法
    print('输出私有方法')
def printme 1(self):            #公共方法
    print('输出公共方法')
    self.__printme_1()
```

```
wz = Site('百度网址', 'www.baidu.com')
wz.printme()           #打印数据
wz.printme 1()         #打印数据，调用公共方法 printme 1()
wz.__printme_1()       #报错，实例不能访问私有方法
```

执行以上代码，输出结果为：

```
name :  百度网址
url :  www.baidu.com
输出公共方法
输出私有方法
Traceback (most recent call last):
  File "D:/Python34/aaa程序/hhh.py", line 16, in <module>
    wz.__printme_1()      # 报错，实例不能访问私有方法
AttributeError: 'Site' object has no attribute '__printme_1'
>>>
```

出错的原因在于，实例不能访问私有方法。

6. 单下划线(_)

以单下划线开始的成员变量叫作保护变量，意思是只有类对象和子类对象自己能访问到这些变量。简单的模块级私有化只需要在属性名前使用一个单下划线字符。以单下划线开头(_singlePrivate)的属性代表不能直接访问的类属性，需要通过类提供的接口进行访问，这样做的目的是防止模块的属性用“from mymodule import *”来加载。

【例 6-5】下划线的使用：

```
class Test():
    def   init   (self):
        pass
    def public(self):
        print ('这是公共方法')
    def  singlePrivate(self):
        print ('这是单下划线方法')
    def   doublePrivate(self):
        print ('这是双下划线方法')
t = Test()
t.public()              #可以调用
t. singlePrivate()      #可以调用
t.__doublePrivate()     #出现错误
```

执行以上代码，输出结果为：

```
这是公共方法
这是单下划线方法
Traceback (most recent call last):
  File "D:/Python34/aaa程序/ex6-5.py", line 14, in <module>
    t.__doublePrivate()      # 出现错误
AttributeError: 'Test' object has no attribute '__doublePrivate'
>>>
```

注意： f._singlePrivate()可以直接访问，不过根据 Python 的约定，应该将其视作 private，而不要在外部使用它们，良好的编程习惯是不要在外部使用它。同时，根据 Python docs 的说明，_object 和__object 的作用域限制在本模块内。

7. 类的专有方法

表 6-1 所示的是 Python 常用的一些专有方法。对于其中的__init__(构造函数)，前面已经介绍，它在生成一个对象时被自动调用。与此相反的是__del__(析构函数)，它在释放一个对象时被自动调用。

表 6-1　类的专有方法

专有方法	说　明
__init__	构造函数，在生成对象时调用
__del__	析构函数，释放对象时使用
__repr__	打印，转换
__setitem__	按照索引赋值
__getitem__	按照索引获取值
__len__	获得长度
__cmp__	比较运算
__call__	函数调用
__add__	加运算
__sub__	减运算
__mul__	乘运算
__div__	除运算
__mod__	求余运算
__pow__	乘方运算
__str__	字符串方法

【例 6-6】 __del__和__repr__专有方法的使用：

```
class Test:
    def   init   (self,name=None):
        self.name = name
    def   del   (self):
        print ("hello world!")
    def   repr   (self):
        return "Study('Jacky')"
    def say(self):
        print (self.name)

Test("Tim")          #自动调用  del  方法
s= Test("Tim")
s.say()
print(s)             #自动调用  repr  方法
print (Test("Kitty"))    #先自动调用__repr__方法，然后自动调用__del__方法
```

执行以上代码，输出结果为：

```
hello world!
Tim
Study('Jacky')
Study('Jacky')
hello world!
>>>
```

输出结果说明：

为何程序输出的第一行是“hello world!”呢？因为执行实例化操作 Test(“Tim”)时，没有把这个实例赋予一个变量，因而这个实例是没有用的，将由垃圾回收器自动回收，当然，回收之前，要释放对象，释放时自动执行析构函数__del__，因而打印出“hello world!”。

第二行的“Tim”由 s.say()输出。执行实例化操作 Test(“Tim”)后会生成一个实例，把该实例赋值给 s 变量，并调用该实例的 say()函数。

第三行输出的是“Study(‘Jacky’)”，这是由 print 函数自动调用__repr__函数输出的。重构__repr__函数后，不管将对象直接输出，还是通过 print 函数输出，都按__repr__函数中定义的格式输出。

第四行的“Study(‘Jacky’)”和第五行的“hello world!”是由 print(Test(“Kitty”))语句输出的。①该语句执行实例化操作 Test(“Kitty”)后，先执行 print 函数，其结果是调用__repr__函数，输出“Study(‘Jacky’)”；②因为执行实例化 Test(“Kitty”)时，没有把这个实例赋予一个变量，所以这个实例是没有用的，将由垃圾回收器自动回收，回收之前先释放对象，此时自动执行析构函数__del__释放该对象，从而打印出“hello world！”。

【例 6-7】__str__专有方法的使用：

```
class Test:
def   init   (self,number):
    self.a=number[0:3]
    self.b=number[3:6]
def   str   (self):
    return "%s %s"%(self.a,self.b)
def test():
num=Test(input("请输入数字: \n"))
print ("输入的数字是:",num)
#执行脚本
test()
```

执行以上代码，输出结果为：

```
请输入数字：
123456
输入的数字是： 123 456
>>>
```

当执行 test()函数时，print()会自动调用__str__函数，从而输出“123 456”(中间带有空格)，说明 print()函数按我们在__str__函数中定义的格式进行了输出。

6.1.3 关于 Python 的作用域和命名空间

类的定义非常巧妙地运用了命名空间，要完全理解接下来的知识，需要先理解作用域和命名空间的工作原理。另外，这些知识对于任何高级 Python 程序员都非常有用。

1. 作用域与命名空间的解释

在 4.2.6 小节中，曾围绕着变量讨论了作用域与命名空间，本节对此进一步讨论。

作用域是指 Python 程序可以直接访问到的命名空间。“直接访问”在这里意味着访问命名空间中的命名时无需加入附加的修饰符。

命名空间本质上是一个字典，它的键就是变量名，它的值就是那些变量的值。Python 使用命名空间来记录变量的轨迹。

在 Python 程序中的任何一个地方，都存在三个可用的命名空间。

(1) 每个函数都有自己的命名空间(称作局部命名空间)，它记录了函数中的变量，包括函数的参数和定义的局部变量。

(2) 每个模块都有自己的命名空间(称作全局命名空间)，它记录了模块中的变量，包括函数、类、其他导入的模块、模块级的变量和常量。

(3) 每个模块都有可访问的内置命名空间，它存放着内建函数和异常。

如果一个命名声明为全局的，那么所有的赋值和引用都直接针对包含模块全局命名的中级作用域。另外，从外部访问到的所有内层作用域的变量都是只读的。

从文本意义上讲，局部作用域引用当前函数的命名空间。在函数之外，局部作用域与全局作用域引用同一命名空间——模块命名空间。而类定义也是局部作用域中的一个命名空间。

作用域决定于源程序的文本：一个定义于某模块中的函数的全局作用域是该模块的命名空间，而不是该函数的别名被定义或调用的位置，了解这一点非常重要。另一方面，命名的实际搜索过程是动态的，是在运行时确定的。然而，Python 语言也在不断发展，以后有可能会成为静态的(编译时确定)，那时就不依赖于动态解析。

2. 命名空间的查找顺序

当一行代码要使用变量 x 的值时，Python 会到所有可用的命名空间去查找变量，其查找顺序如下。

(1) 局部命名空间——特指当前函数或类的方法。如果函数中定义了一个局部变量 x，Python 将使用这个变量，然后停止搜索。

(2) 全局命名空间——特指当前的模块。如果模块中定义了一个名为 x 的变量、函数或类，Python 将使用它，然后停止搜索。

(3) 内置命名空间——对每个模块都是全局的，作为最后的尝试，Python 将假设 x 是内建函数或变量。

(4) 如果 Python 在这些命名空间中都找不到 x，它将放弃查找并引发一个 NameError 异常，同时传递 There is no variable named ‘x’这样一条信息。

不同的命名空间在不同的时刻创建，有不同的生存期。包含内建函数的命名空间在 Python 解释器启动时创建，会一直保留，不被删除。模块的全局命名空间在模块定义被读入时创建，通常，模块命名空间也会一直保存到解释器退出。由解释器在最高层调用执行的语句，不管它是从脚本文件中读入还是来自交互式输入，都是__main__模块的一部分，所以它们也拥有自己的命名空间。内置命名也同样被包含在一个模块中，它被称作

__builtin__，该模块包含内建函数、异常以及其他属性。当函数被调用时，创建一个局部命名空间，我们可以通过 globals()和 locals()两个内建函数判断某一名字属于哪个名称空间，locals()是只读的，globals()不是。

【例 6-8】locals()函数示例：

```
def foo(arg, a):
   x = 1
   y = 'abc'
   for i in range(5):
       j = 2
       k = i
   print (locals())
#调用函数的打印结果
foo(2,3)
```

执行以上代码，输出结果为：

```
{'a': 3, 'arg': 2, 'j': 2, 'i': 4, 'k': 4, 'x': 1, 'y': 'abc'}
>>>
```

locals()实际上没有返回局部命名空间，它返回的是局部命名空间的一个拷贝，所以对它进行改变时，对局部命名空间中的变量值并无影响。

【例 6-9】globals()函数示例：

```
print("当前的全局命名空间:")
var=globals()
print(var)
```

执行以上代码，输出结果为：

```
当前的全局命名空间:
{'__loader__': <class '_frozen_importlib.BuiltinImporter'>, 'var': {...}, '__
package__': None, '__spec__': None, '__file__': 'D:/Python34/aaa程序/ex6-9.py'
, '__doc__': None, '__name__': '__main__', '__builtins__': <module 'builtins'
(built-in)>}
>>>
```

globals()函数返回实际的全局命名空间，而不是它的一个拷贝。所以对 globals()函数所返回的 var 的任何改动都会直接影响到全局变量。

3. 嵌套函数命名空间的查找步骤

(1) 先在当前函数(嵌套的函数或 lambda 匿名函数)的命名空间中搜索。

(2) 然后在父函数的命名空间中搜索。

(3) 接着在模块命名空间中搜索。

(4) 最后在内建函数的命名空间中搜索。

下面举例说明。

【例 6-10】嵌套函数命名空间示例：

```
address = "地址: "
def func country(country):
    def func part(part):
        city= "天津 "    #覆盖父函数的 part 变量
        print(address + country + city + part)
```

```
    city = "北京 "       #初始化 city 变量
    #调用内部函数
    func part("西青")     #初始化 part 变量
#调用外部函数
func_country("中国 ")     #初始化 country 变量
```

执行以上代码，输出结果为：

```
地址：中国 天津 西青
>>>
```

以上例子中，address 在全局命名空间中，country 在父函数的命名空间中，city 和 part 在子函数的命名空间中。

6.2　Python 类与对象

6.2.1　类对象

类对象支持两种操作：属性引用和实例化。

类对象的属性引用与 Python 中所有属性的引用一样，都使用标准的语法：obj.name。类对象创建之后，类命名空间中所有的命名都是有效属性名。在 Python 中，方法定义在类的定义(即声明)中，但只能被类对象的实例所调用。调用一个方法的途径分三步。

(1) 定义类和类中的方法。

(2) 创建一个或若干个实例，即将类实例化。

(3) 用所创建的实例调用方法。

【例 6-11】类的定义与实例化：

```
class MyClass:
    """一个简单的类实例"""
    i = 12
    def f(self):
        return 'hello world'
#实例化类
MyClass1 = MyClass()
#访问类的属性和方法
print("MyClass 类的属性 i 为：", MyClass1.i)
print("MyClass 类的方法 f 输出为：", MyClass1.f())
```

执行以上代码，输出结果为：

```
MyClass 类的属性 i 为： 12
MyClass 类的方法 f 输出为： hello world
>>>
```

本例中，MyClass1.i 和 MyClass1.f()都是有效的属性引用，分别返回一个整数和一个方法对象。也可以对类属性赋值，即可以通过给 MyClass1.i 赋值来修改它。如：

```
MyClass1.i=56
print("修改后 MyClass 类的属性 i 为：", MyClass1.i)
```

执行以上代码，输出结果为：

```
修改后MyClass 类的属性 i 为: 56
>>>
```

类的实例化使用函数符号，只要将类对象看作是一个返回新的类实例的无参数函数即可。例如(假设沿用前面的类)：

```
MyClass1 = MyClass()
```

该语句创建了一个新的类实例(对象)，并将该对象赋给局部变量 MyClass1。通过实例化操作(“调用” 一个类对象)来创建一个空的对象时，通常会把这个新建的实例赋给一个变量。赋值在语法上不是必需的，但如果不把这个实例保存到一个变量中，它就没有用，会被垃圾收集器自动回收，因为没有任何引用指向这个实例。换言之，刚刚所做的一切，仅仅是为那个实例分配了一块内存，随即又释放掉，这样做是没有任何意义的。上一节的例 6-6 程序中第一条执行语句 Test(“Tim”)就是一个很典型的例子。

很多类都倾向于创建一个有初始化状态的对象。因此类可能会定义一个名为__init__()的特殊方法(称为构造方法，前面已提到)，像下面这样：

```
def   init   (self):
    self.data = []
```

当类被调用时，实例化的第一步就是创建实例对象，一旦对象创建，Python 就检查是否已经实现__init__()方法。默认情况下，如果没有定义(或覆盖)特殊方法__init__()，对实例不会施加任何特别的操作。任何所需的特定操作，都需要程序员实现__init__()方法，覆盖它的默认行为。所以在下例中，可以这样创建一个新的实例：

```
MyClass_1 = MyClass()
```

当然，出于弹性的需要，__init__()方法可以有参数。事实上，通过__init__()方法参数被传递到类的实例上。

【例 6-12】使用带参数的__init__()方法初始化：

```
class Complex:
     def   init   (self, realpart, imagpart):
         self.r = realpart
         self.i = imagpart
x = Complex(2.4, -4.6)
print(x.r, x.i)
```

执行以上代码，输出结果为：

```
2.4 -4.6
>>>
```

6.2.2 类的属性

有两种有效的属性名：数据属性和特殊类属性。

1. 数据属性

这相当于 Smalltalk 中的实例变量或 C++中的数据成员。与局部变量一样，数据属性不需要声明，第一次使用时它们就会生成。

【例 6-13】类数据属性说明：

```
class foo(object):
    f = 100
print (foo.f)
print (foo.f+1)
```

执行以上代码，输出结果为：

```
100
101
>>>
```

2. 特殊类属性

对任何类 foo，其部分特殊属性如表 6-2 所示。

表 6-2　特殊类属性

类 属 性	类属性说明
foo.__name__	类 foo 的名字(字符串)
foo.__doc__	类 foo 的文档字符串
foo.__bases__	类 foo 的所有父类构成的元组
foo.__dict__	类 foo 的属性
foo.__module__	类 foo 定义所在的模块
foo.__class__	实例 foo 对应的类

【例 6-14】dir()函数和类属性__dict__的使用：

```
class MyClass(object):
    'MyClass 类定义'
    myVer = '3.4'
    def showMyVer (self):
        print (MyClass.myVer)
print (dir(MyClass))
print (MyClass.__dict__)
```

根据上面定义的类，我们使用 dir()和特殊类属性__dict__来查看一下类的属性。

执行以上代码，输出结果为：

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__f
ormat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__
le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'myVer', 'showMyVer']
{'__module__': '__main__', '__dict__': <attribute '__dict__' of 'MyClass' obj
ects>, '__weakref__': <attribute '__weakref__' of 'MyClass' objects>, '__doc_
_': 'MyClass 类定义', 'showMyVer': <function MyClass.showMyVer at 0x02B62150>,
'myVer': '3.4'}
>>>
```

从上面可以看到，dir()返回的仅是对象属性的一个名字列表，而__dict__返回的是一个字典，它的键是属性名，值是相应属性的数据值。结果还显示 MyClass 类中两个熟悉的属性 showMyVer 和 myVer，以及一些新的属性。

- __dict__属性包含一个字典，由类的数据属性组成。访问一个类属性的时候，Python 解释器将会搜索字典以得到需要的属性。如果在__dict__中没有找到，将会在基类的字典中进行搜索，采用“深度优先搜索”顺序。基类集的搜索是按顺序的，从左到右，按其在类定义时，定义父类参数的顺序。对类的修改仅会影响到此类的字典，基类的__dict__属性不会被改动。
- __name__是给定类的字符名字。它适用于那种只需要字符串(类对象的名字)，而非类对象本身的情况。

  ```
  print (MyClass.__name__)
  ```

  输出：`MyClass`
- __doc__是类的文档字符串，与函数及模块的文档字符串相似，必须紧随头行(header line)。文档字符串不能被派生类继承，也就是说，派生类必须含有它们自己的文档字符串。

  ```
  print (MyClass.__doc__)
  ```

  输出：`MyClass 类定义`
- __module__的引入是为了更清晰地对类进行描述，这样类名就完全由模块名所限定了。

  ```
  print (MyClass.__module__)
  ```

  输出：`__main__`

6.2.3 实例属性

内建函数 dir()可以显示类属性，同样还可以打印所有的实例属性。

【例 6-15】实例属性说明：

```
class foo(object):
   pass
foo_1 = foo()
print (dir(foo_1))
```

执行以上代码，输出结果为：

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__f
ormat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__
le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
>>>
```

实例有两个特殊属性，如表 6-3 所示。

表 6-3　特殊实例属性

实例属性	说　明
foo_1.__class__	实例化 foo_1 的类
foo_1.__dict__	foo_1 的属性

现在使用类 foo 及其实例 foo_1 来看看这些特殊实例属性。

【例 6-16】查看实例属性__dict__和__class__：

```
class foo(object):
   pass
foo_1 = foo()
print (foo_1.__dict__)
print (foo_1.__class__)
```

执行以上代码，输出结果为：

```
{}
<class '__main__.foo'>
>>>
```

foo_1 现在还没有数据属性，但我们可以添加一些，再来检查__dict__属性。

【例 6-17】查看类的数据属性：

```
class foo(object):
   pass
foo_1 = foo()
foo_1.f = 100
foo_1.b = "hello"
print (foo_1.__dict__)
print (foo_1.__class__)
```

执行以上代码，输出结果为：

```
{'b': 'hello', 'f': 100}
<class '__main__.foo'>
>>>
```

注意： __dict__属性由一个字典组成，包含一个实例的所有属性。键是属性名，值是属性相应的数据值。字典中仅有实例属性，没有类属性或特殊属性。

6.2.4　一些说明

同名的数据属性会覆盖方法属性，最好以某种命名约定来避免冲突，因为这在大型程序中可能会导致出现难以发现的 bug。可选的约定包括：

- 类名用大写字母书写。
- 方法的首字母大写。
- 数据属性名前缀小写(可能只是一个下划线)。
- 方法使用动词而数据属性使用名词。

数据属性可以由方法引用，也可以由普通用户调用。换句话说，类不能实现纯的数据类型。事实上，Python 中没有什么办法可以强制隐藏数据，一切都基于约定的惯例。

Python 程序员应该小心使用数据属性，因为随意修改数据属性可能会破坏本来由方法维护的数据一致性。需要注意的是，程序员只要注意避免命名冲突，就可以随意向实例中添加数据属性而不会影响方法的有效性。再次强调，命名约定可以省去很多麻烦，如果不遵守这个约定，其他的 Python 程序员阅读你的代码时会感到不便。

从方法内部引用数据属性以及其他方法没有什么快捷的方式。这事实上增加了方法的可读性：即使粗略地浏览一个方法，也不会有混淆局部变量和实例变量的机会。如前所

述，方法的第一个参数命名为 self，这仅仅是一个约定，对 Python 而言，self 绝对没有任何特殊含义。通过使用 self 参数的方法属性，方法可以调用其他的方法。

【例 6-18】self 参数的使用：

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
        print(self.data)
    def addtwice(self, x):
        self.add(x)
        print(self.data)
bag_1 = Bag()
bag_1.add(5)
bag_1.addtwice(6)
```

执行以上代码，输出结果为：

```
[5]
[5, 6]
[5, 6]
>>>
```

6.3 继　　承

6.3.1 单继承

Python 同样支持类的继承。如果一种语言不支持继承，类就没有什么意义。类还允许派生，即用户可以创建一个子类(又称派生类)，它也是类，而且继承父类(即基类)的所有特征和属性。

创建派生类的语法格式为：

```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

基类名 BaseClassName 必须与派生类定义在一个作用域内。除了用类名，还可以用表达式。基类可以定义在另一个模块中，这一点非常有用，语法格式如下：

```
class DerivedClassName(modname.BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

【例 6-19】单继承举例：

```
#类定义
class people:
    #定义基本属性
```

```
    name = ' '
    age = 0
    #定义私有属性，私有属性在类外部无法直接进行访问
      weight = 0
    #定义构造方法
    def   init   (self,n,a,w):
        self.name = n
        self.age = a
        self.  weight = w
    def speak(self):
         print("%s says: I am %d years old" %(self.name,self.age))
#单继承
class student(people):
    grade = ' '
    def   init   (self,n,a,w,g):
        #调用父类的构造函数
        people.  init   (self,n,a,w)
        self.grade = g
    #重写父类的方法
    def speak(self):
       print("%s says: I am %d years old, I am in Grade %d."
       %(self.name,self.age,self.grade))
s = student('Tom',10,90,3)
s.speak()
```

执行以上代码，输出结果为：

```
Tom says: I am 10 years old, I am in Grade 3.
>>>
```

提示：

- 派生类定义的执行过程和基类是一样的。构造派生类对象时就继承了基类。这在解析属性引用的时候尤其有用：如果在类中找不到请求调用的属性，就搜索基类。如果基类是由别的类派生而来，这个规则会递归地应用上去。
- 派生类的实例化没有什么特殊之处：DerivedClassName()(示列中的派生类)创建一个新的类实例。方法引用按如下规则解析：搜索对应的类属性，必要时沿基类链逐级搜索，如果找到函数对象，这个方法引用就是合法的。
- 派生类可能会覆盖其基类的方法。因为方法调用同一个对象中的其他方法时没有特权，基类的方法调用同一个基类的方法时，可能实际上最终调用派生类中的覆盖方法。对于 C++程序员来说，Python 中的所有方法本质上都是虚方法。
- 派生类中的覆盖方法可能是想要扩充而不是简单替代基类中的重名方法。有个简单的方法可直接调用基类方法，即 BaseClassName.methodname(self, arguments)，有时这对程序员也很有用。要注意的是，只有基类在同一全局作用域定义或导入时才能这样用。

6.3.2　多继承

Python 同样有限地支持多继承形式。多继承的类定义语法格式如下：

```
class DerivedClassName(Base1, Base2, Base3):
```

```
    <statement-1>
    ...
    <statement-N>
```

这里唯一需要解释的语义是解析类属性的规则。顺序是深度优先，从左到右。因此，如果在 DerivedClassName(示例中的派生类)中没有找到某个属性，就会搜索 Base1，然后递归地搜索其基类，如果最终没有找到，就搜索 Base2，依此类推。深度优先不区分属性继承自基类还是直接定义。

【例 6-20】多继承举例：

```
#类定义
class people:
    #定义基本属性
    name = ' '
    age = 0
    #定义私有属性，私有属性在类外部无法直接进行访问
      weight = 0
    #定义构造方法
    def   init   (self,n,a,w):
        self.name = n
        self.age = a
        self.   weight = w
    def speak(self):
        print("%s says: I am %d years old" %(self.name,self.age))
#单继承示例
class student(people):
    grade = ' '
    def   init   (self,n,a,w,g):
        #调用父类的构函
        people.  init  (self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s says: I am %d years old, I am in Grade %d."
%(self.name,self.age,self.grade))
#另一个类，多重继承之前的准备
class speaker():
    topic = ' '
    name = ' '
    def   init   (self,n,t):
        self.name = n
        self.topic = t
    def speak(self):
        print("I am %s,I am a speaker,my topic
is %s"%(self.name,self.topic))
#多重继承
class sample(speaker,student):
    a =' '
    def   init   (self,n,a,w,g,t):
        student.__init__(self,n,a,w,g)
```

```
        speaker.__init__(self,n,t)

test_1 = sample("Tom",12,90,3,"One World One Dream")
test_1.speak()    #方法名相同，默认调用的是在括号中排前的父类的方法，
                  #即 speaker 类里面的方法
```

执行以上代码，输出结果为：

```
I am Tom,I am a speaker,my topic is One World One Dream
>>>
```

注意： 不加限制地使用多继承会带来维护上的噩梦，因为 Python 中只依靠约定来避免命名冲突。多继承的一个很有名的问题是，派生继承的两个基类都是从同一个基类继承而来。目前还不清楚这在语义上有什么意义，然而，这会造成意想不到的后果。

6.3.3　补充

上述例子中展示了方法重写的概念，下面再展示一个简单的方法重写的例子，从而强调方法重写的重要性。

1. 方法重写

如果父类方法的功能不能满足需求，可以在子类里重写父类的方法。

【例 6-21】方法重写：

```
class Parent:                  #定义父类
   def myMethod(self):
      print ('调用父类方法')
class Child(Parent):              #定义子类
   def myMethod(self):
      print ('调用子类方法')
Child_1 = Child()               #子类实例
Child_1.myMethod()             #子类调用重写方法
```

执行以上代码，输出结果为：

```
调用子类方法
>>>
```

2. 运算符重载

运算符重载是针对新类型数据的实际需要，对原有运算符进行适当的改造。一般来说，重载的功能应当与原有功能相类似，不能改变原运算符的操作对象个数，同时至少要有一个操作对象是自定义类型。

Python 同样支持运算符重载，它的运算符重载就是通过重写这些 Python 内建方法来实现的。这些内建方法都是以双下划线开头和结尾的(类似于__X__的形式)，Python 通过这种特殊的命名方式来拦截操作符，以实现重载。当 Python 的内置操作运用于类对象时，Python 会去搜索并调用对象中指定的方法来完成操作。运算符重载是通过创建运算符函数实现的，运算符重载实际是一个函数，所以运算符的重载实际上是函数的重载。解释程序对运算符重载的选择遵循着函数重载的选择原则，当遇到不很明显的运算时，编译程序将

去寻找参数相匹配的运算符函数。

类可以重载加减运算、打印、函数调用、索引等内置运算，运算符重载使我们的对象的行为与内置对象的行为一样。

例如，如果类实现了__add__方法，当类的对象出现在“+”运算符中时，会调用这个方法，如果类实现了__sub__方法，当类的对象出现在“-”运算符中时，会调用这个方法。重载这两个方法就可以在普通的类对象上添加“+”或“-”运算。

下面的代码演示了如何使用“+”和“-”运算符。

【**例 6-22**】运算符重载：

```
class Computation():
   def   init   (self,value):
      self.value = value
   def   add   (self,other):
      return self.value + other.value
   def   sub   (self,other):
      return self.value - other.value

c1 = Computation(10)
c2 = Computation(10)
print(c1 + c2)  #“+”在作用于类对象，实现加法运算符的重载
print(c1 - c2)  #“-”在作用于类对象，实现减法运算符的重载
```

执行以上代码，输出结果为：

```
20
0
>>>
```

注意：

- 本例通过重载__add__、__sub__两个方法实现对加法、减法两个运算符的重载。
- 重载之后，运算符的优先级和结合性都不会改变。
- 在所有重载方法的名称前后都有两个下划线，以便与同类中所定义的变量名区别开来。
- 运算符重载方法都是可选的，如果没有编写或继承一个方法，该类直接不支持这些运算，并且试图使用它们时会引发一个异常。类可重载所有的 Python 表达式运算符，并且使类实例的行为像内置类型。
- 运算符重载只是意味着在类方法中拦截内置的操作，若类的实例出现在内置操作中，则 Python 自动调用重载的方法，并且重载方法的返回值变成了相应操作的结果，如本例所示。

3. 空类的使用

有时候，类似于 Pascal 中“记录(record)”或 C 中“结构体(struct)”的数据类型很有用，因为它可以将一组已命名的数据项绑定在一起。一个空的类定义可以很好地实现它，如下面的例子。

【**例 6-23**】pass 的使用：

```
class Employee:
   pass
```

```
john = Employee()        #创建空的类对象
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
print(john.name)
print(john.dept)
print(john.salary)
```

执行以上代码，输出结果为：

```
John Doe
computer lab
1000
>>>
```

注意： 某一段 Python 代码如果需要一个特殊的抽象数据结构的话，通常可以传入一个类，事实上，这模仿了该类的方法。例如，如果有一个用于从文件对象中格式化数据的函数，可以定义一个带有 read()和 readline()方法的类，依次从字符串缓冲区读取数据，然后将该类的对象作为参数传入前述的函数。

4. if __name__ = '__main__' 的作用

前已提及，这条语句的意思是，让程序员写的脚本模块既可以导入到别的模块中调用，也可以在该模块中自己执行。

【例 6-24】定义一个模块，名字为 ylhtext.py：

```
#ylhtext.py
def main():
   print ("hello world,I am in %s now." %  name  )
if   name   == '  main  ':
   main()
```

执行以上代码，输出结果为：

```
hello world,I am in __main__ now.
>>>
```

在这个文件中定义了一个 main()函数，如果执行该.py 文件，则 if 语句中的内容被执行，成功调用 main()函数。现在我们从另一个模块导入该模块，这个模块名字是 test1.py。

【例 6-25】模块导入：

```
#test1.py
from ylhtext import main
main()
```

执行以上代码，输出结果为：

```
hello world,I am in ylhtext now.
>>>
```

注意： if __name__ = '__main__' 下面的 main()函数没有执行，这样一来，既可以让模块文件独立运行，也可以被其他模块导入，而且不会执行函数两次。如果独立运行某个.py 文件，在该文件中"__name__ == '__main__'"是 True；但如果直接从另外一个.py 文件通过 import 导入该文件，则此时__name__的值就是这个.py 文件的名字，而不是__main__。

6.3.4 isinstance 函数

isinstance()函数是 Python 语言的一个内建函数。

语法：isinstance(object, type)

作用：判断一个对象或者变量是否是一个已知的类型。

(1) 针对类来说，如果参数 object 是 type 类的实例或者 object 是 type 类的子类的一个实例，则返回 True。如果 object 不是一个给定类型的对象，则返回结果总是 False。

【例 6-26】使用 isinstance()函数判断一个对象是否是已知的类型：

```
class objA:
   pass
class objB(objA):
   pass
A = objA()
B = objB()
print (isinstance (A, objA))
print (isinstance (B, objA))
print (isinstance (A, objB))
print (isinstance (B, objB))
```

执行以上代码，输出结果为：

```
True
True
False
True
>>>
```

(2) 针对变量来说，其第一个参数(object)为变量，第二个参数(type)为类型名(如 int)或由类型名组成的一个元组(如(int,list,float))，其返回值为布尔型(True 或 False)。若第二个参数为一个元组，则变量类型与元组中的类型名之一相同时即返回 True。

【例 6-27】使用 isinstance()函数判断变量 a 是否是一个已知的类型：

```
a = 2
print (isinstance (a,int))
print (isinstance (a,str))
print (isinstance (a,(str,int,list)))
```

执行以上代码，输出结果为：

```
True
False
True
>>>
```

【例 6-28】使用 isinstance()函数判断字符串变量是否是一个已知的类型：

```
a = "b"
print (isinstance (a,str))
print (isinstance (a,int))
print (isinstance (a,(int,list,float)))
print (isinstance (a,(int,list,float,str)))
```

执行以上代码，输出结果为：

```
True
False
False
True
>>>
```

6.3.5　super()函数

当存在继承关系的时候，有时候需要在子类中调用父类的方法。如果修改父类名称，那么在子类中会涉及多处修改。另外，Python 是允许多继承的语言，子类调用的方法在多继承时就需要重复写多次，显得累赘。为了解决这些问题，Python 引入了 super()机制，语法格式如下：

```
super(type[, object-or-type])
```

【例 6-29】super()函数在无参数类对象中的使用：

```
class A(object):
    def   init   (self):
        print ("enter A")
        print ("leave A")
class BA. :             #B 继承 A
    def   init   (self):
        print ("enter B")
        super(B, self).  init   ()
        print ("leave B")
b = B()
```

执行以上代码，输出结果为：

```
enter B
enter A
leave A
leave B
>>>
```

注意： 在 Python 2 中，super()是一定要有参数的。Python 2 对 super(B, self).__init__()是这样理解的：super(B, self)首先找到 B 的父类(就是类 A)，然后把类 B 的对象 self 转换为类 A 的对象，然后被转换的类 A 对象调用自己的__init__函数。Python3.x 可以不加参数而直接写成 super().__init__()，并且可以向父类中的属性赋值。

【例 6-30】super()函数在带参数类对象中的使用：

```
class Foo(object):
  def   init   (self, a, b):
    self.a = a
    self.b = b
class Bar(Foo):
  def   init   (self, a, c):
    super().  init   (a,34)
    self.c = c
```

```
n = Bar("hello","world")
print (n.a)
print (n.b)
print (n.c)
```

执行以上代码，输出结果为：

```
hello
34
world
>>>
```

6.4　案例实训：Python 面向对象编程案例演练

本节以几个典型的案例展示 Python 面向对象的编程思想。

1. 栈

栈(stack)，亦称堆栈，是一种“先进后出”或者“后进先出”的装载数据的方式，这里的数据可以是数字、字母、字符串等。

进栈方式如图 6-1 所示，出栈方式如图 6-2 所示。

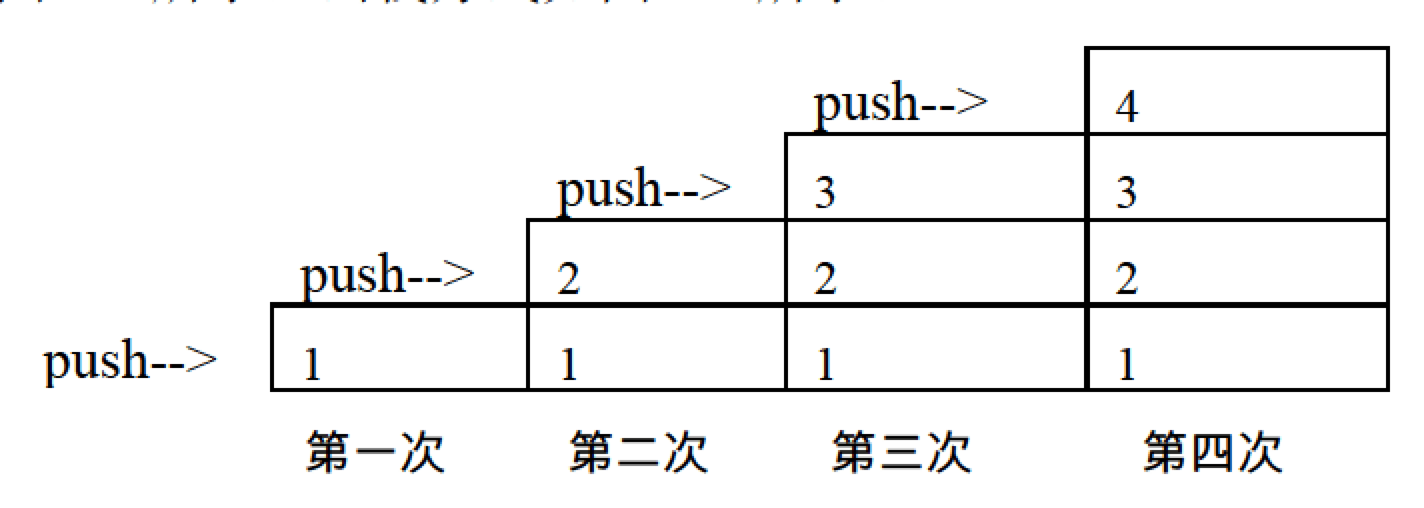


图 6-1　四次进栈 push 操作说明

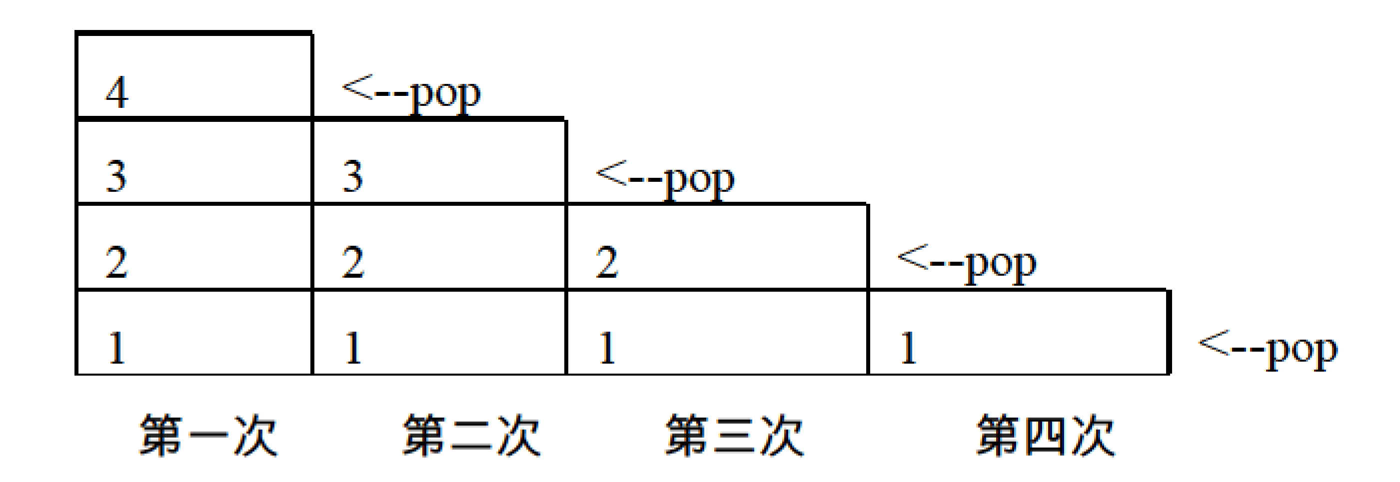


图 6-2　四次出栈 pop 操作说明

栈常见的操作如下。

stack()：建立一个空的栈对象，调用__init__()方法。

push()：把一个元素添加到栈的最顶层。

pop()：删除栈最顶层的元素，并返回这个元素。

isfull()：判断栈是否已满，栈满返回 True，不满返回 False。

isempty()：判断栈是否为空，是空返回 True，不空返回 False。

peek()：返回栈顶元素，并不删除它。

这里使用 Python 的 list 对象模拟栈的实现，具体代码如下：

```
class Stack():
    """用列表模拟栈"""
    def __init__(self,size):
        self.size=size
        self.stack=[]
        self.top=-1
    def push(self,ele):      #入栈之前检查栈是否已满
        if self.isfull():
            print("out of range")
            return
        else:
            self.stack.append(ele)
            self.top+=1
    def pop(self):         #出栈之前检查栈是否为空
        if self.isempty():
            print("stack is empty")
            return
        else:
            self.top-=1
            return self.stack.pop()
    def isfull(self):
        return self.top+1==self.size
    def isempty(self):
        return self.top==-1
    def peek(self):
        if not self.isempty():
          return self.stack[len(self.stack)-1]
s=Stack(20)
print (s.pop())
for i in range(10):
    s.push(i)
print (s.pop())
print (s.pop())
print (s.pop())
print (s.pop())
print (s.isempty())
print (s.isfull())
print (s.peek())
```

执行以上代码，输出结果为：

```
stack is empty
None
9
8
7
6
False
False
5
>>>
```

说明如下。

(1) 本程序由两部分组成，前半部分是栈类的定义与栈的各种操作方法的实现，后半部分是类的实例化与使用。

(2) 实例化后，先检测栈是否为空，若为空，则输出提示信息“stack is empty”，然后用 for 循环依次将 10 个整数入栈，并对入栈的数据进行操作。

注意： 用列表来模拟栈，并将栈的方法封装在类中，在类实例化后，直接用实例对象调用类中的方法，而不必在意类方法的实现细节，同时实现了数据与方法的分离。这就是面向对象的编程思想，读者从中不难看出面向对象编程的本质——隐藏了程序实现的细节，实现了静态数据与动态功能的分离。

2. 队列

队列(queue)是一种“先进先出”或者“后进后出”的装载数据方式，这里的数据可以是数字、字母、字符串等。进队列方式如图 6-3 所示，出队列方式如图 6-4 所示。

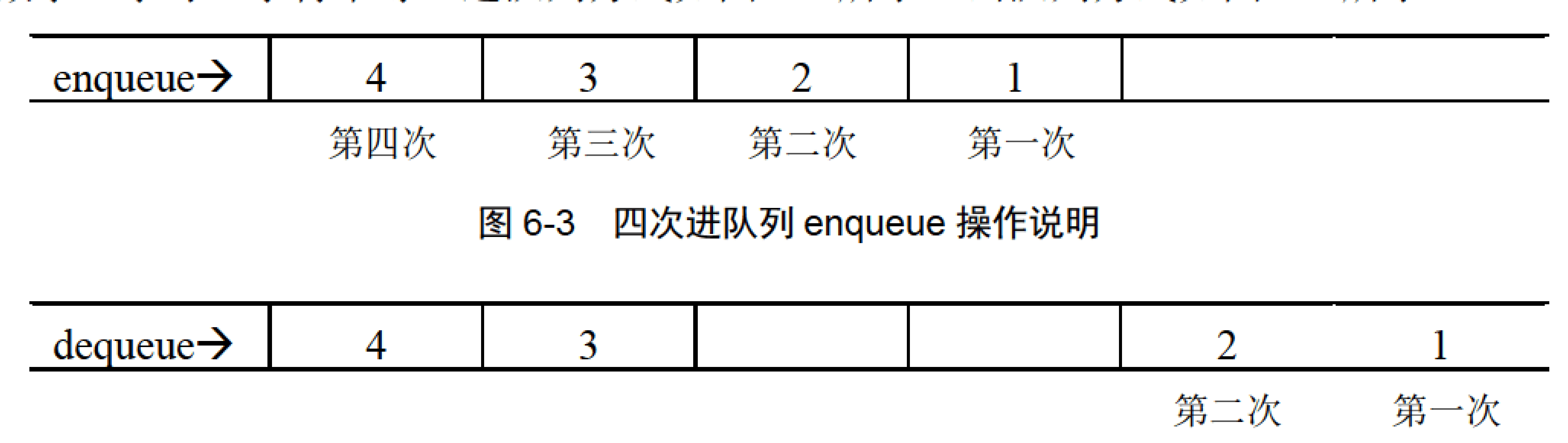


图 6-3　四次进队列 enqueue 操作说明

图 6-4　两次出队列 enqueue 操作说明

队列常见的操作如下。

queue()：建立一个空的队列对象，调用__init__()方法。

enqueue()：把一个元素添加到当前队列元素的最后位置。

dequeue()：删除队列最前面的元素，并返回这个元素。

isfull()：判断队列是否已满，队列满返回 True，不满返回 False。

isempty()：判断队列是否为空，队列为空返回 True，不空返回 False。

这里使用 Python 的 list 对象模拟队列的实现，具体代码如下：

```
class Queue():
"""用列表模拟队列"""
    def __init__(self,size):
        self.size=size
        self.front=-1
        self.rear=-1
        self.queue=[]
    def enqueue(self,ele):    #入队操作
        if self.isfull():
            print("queue is full")
            return
        else:
            self.queue.append(ele)
```

```
            self.rear+=1
    def dequeue(self):      #出队操作
        if self.isempty():
            print("queue is empty")
            return
        else:
            self.front+=1
            return self.queue[self.front]
    def isfull(self):
        return self.rear-self.front+1==self.size
    def isempty(self):
        return self.front==self.rear
q=Queue(15)
for i in range(10):
    q.enqueue(i)
print (q.dequeue())
print (q.dequeue())
print (q.dequeue())
print (q.dequeue())
print (q.dequeue())
print (q.isempty())
print (q.isfull())
```

执行以上代码，输出结果为：

```
0
1
2
3
4
False
False
>>>
```

说明如下。

(1) 本程序与前一个程序类似，也由两部分组成，前半部分是队列类的定义与队列各种操作方法的实现，后半部分是类的实例化与使用。

(2) 实例化之后，使用 for 循环依次将 10 个整数放入队列，并且对队列中的数据进行操作。

注意：

- 本程序也是使用列表来模拟队列的，操作队列的方法被封装在类中，在类实例化后，直接用实例对象调用类中的方法，从而隐藏了程序实现的细节，实现了数据与方法的分离，也体现了抽象程序设计的风格。
- 上面的两个例子虽然简单，但“麻雀虽小，五脏俱全”，它们体现了面向对象编程的思想，展示了面向对象编程的优势。读者在以后的编程实践中，应尽量多地使用面向对象的编程思想，运用面向对象技术来解决实际问题。

3. 中文分词的最大正向匹配算法

中文分词一直都是中文自然语言处理领域的基础研究。中文分词的一个最基础算法是最大匹配算法(Maximum Matching，MM)。MM 算法有两种：最大正向匹配和最大逆向匹

配，这里介绍最大正向匹配算法。这是一种基于字典的匹配方法，从左向右扫描，寻找词的最大匹配。首先规定一个词的最大长度，每次扫描时寻找当前开始的这个长度的词，与字典中的词匹配，如果没找到，就缩短长度继续寻找，直到找到或者成为单字。算法流程如图 6-5 所示。

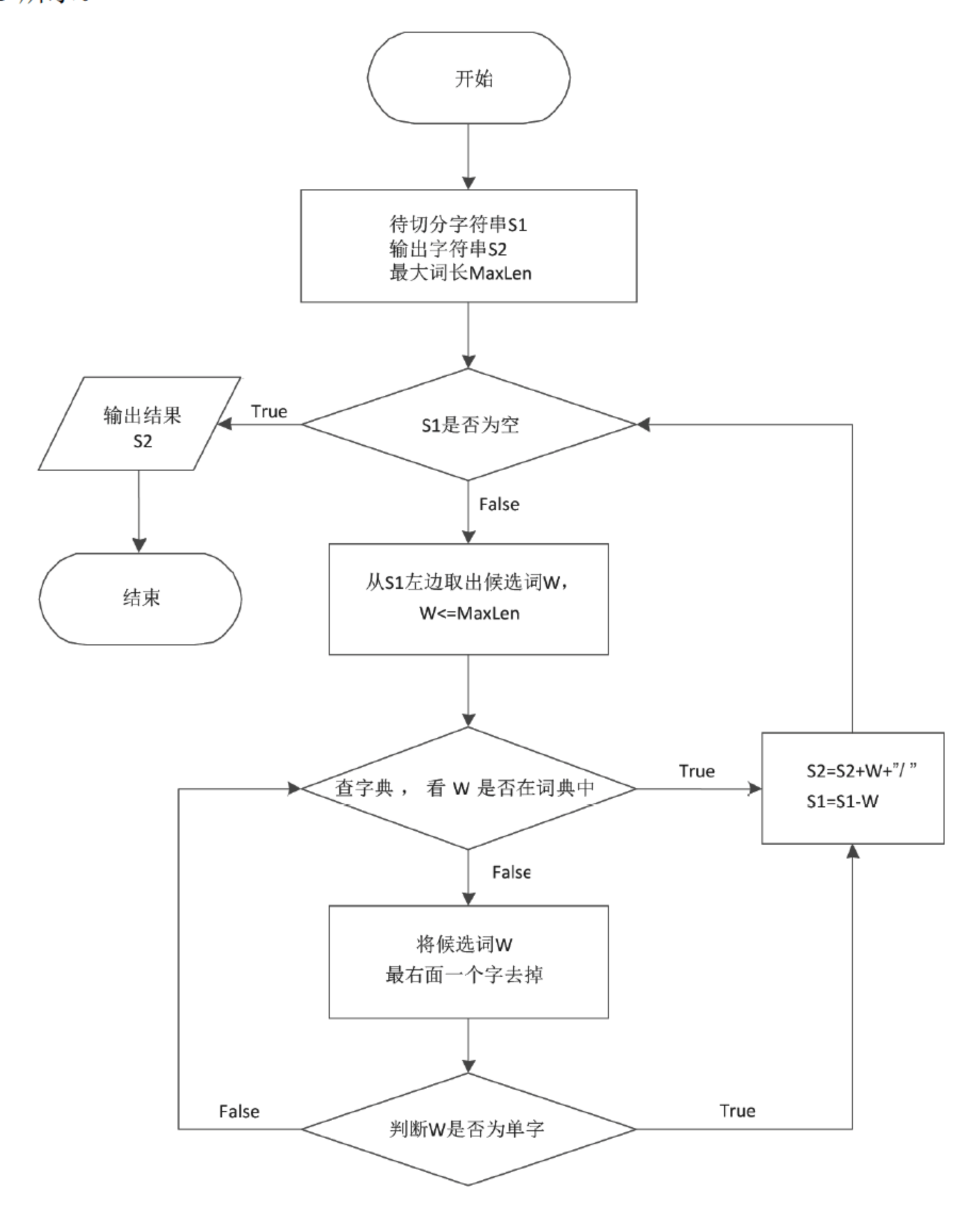


图 6-5　最大正向匹配算法流程图

算法解释：

S1="中文分词是汉语言处理的基础"，设定最大词长 MaxLen = 5，S2= ""。

字典中含有三个词：[中文分词]、[汉语言处理]、[基础]。

(1)　S2=""；S1 不为空，从 S1 左边取出候选子串 W="中文分词是"。

①　查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="中文分词"。

②　查词表，W 在词表中，将 W 加入到 S2 中，S2="中文分词/ "，并将 W 从 S1 中去

掉，此时 S1="是汉语言处理的基础"。

(2) S1 不为空，从 S1 左边取出候选子串 W="是汉语言处"。

① 查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="是汉语言"。

② 查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="是汉语"。

③ 查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="是汉"。

④ 查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="是"，这时 W 是单字，将 W 加入到 S2 中，S2="中文分词/ 是/ "，并将 W 从 S1 中去掉，此时 S1="汉语言处理的基础"。

(3) S1 不为空，从 S1 左边取出候选子串 W="汉语言处理"。

查词表，W 在词表中，将 W 加入到 S2 中，S2="中文分词/ 是/ 汉语言处理/ "，并将 W 从 S1 中去掉，此时 S1="的基础"。

(4) S1 不为空，从 S1 左边取出候选子串 W="的基础"。

① 查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="的基"。

② 查词表，W 不在词表中，将 W 最右边一个字去掉，得到 W="的"；这时 W 是单字，将 W 加入到 S2 中，S2="中文分词/ 是/ 汉语言处理/ 的/ "，并将 W 从 S1 中去掉，此时 S1="基础"。

(5) S1 不为空，从 S1 左边取出候选子串 W="基础"。

查词表，W 在词表中，将 W 加入到 S2 中，S2="中文分词/ 是/ 汉语言处理/ 的/ 基础/ "， 并将 W 从 S1 中去掉，此时 S1=""。

(6) S1 为空，输出 S2="中文分词/ 是/ 汉语言处理/ 的/ 基础/ "作为分词结果，分词过程结束。

算法实现：

该算法接受两个参数：输入字符串和词典，并且自动计算最大词长：

```
class FMM:
    """最大正向匹配算法类"""
    size = 0    #最大词长
    words = []  #存放分词结果的列表
    index = 0   #当前的待分词的位置
    def   init  (self,s,w,i):
        self.size = s
        self.words = w
        self.index = i

    def search maxLen(self,dic):
        """寻找最大词长"""
        for i in dic:
            if len(i)>self.size:
                self.size=len(i)
        return

    def max match segment(self,chars,dic):
        """最大正向匹配算法"""
        while self.index < len(chars):
```

```
            matched = False
            for i in range(self.size, 0, -1):
                string = chars[self.index:self.index+i]
                if string in dic:
                    self.words.append(string)
                    self.words.append("/")
                    matched = True
                    break
            if not matched:
                i = 1
                self.words.append(chars[self.index])
                self.words.append("/")
            self.index += i
        print (self.words)
        return
if   name   =="  main  ":
    dictionary = {"脚本语言" , "可以","一种","应用","计算机","领域","很多"}
    string1 = "脚本语言可以应用在计算机的很多领域"
    fmm1 = FMM(1,[],0)
    fmm1.search maxLen(dictionary)
fmm1.max_match_segment(string1,dictionary)
```

执行以上代码，输出结果为：

```
['脚本语言', '/', '可以', '/', '应用', '/', '在', '/', '计算机', '/', '的', '/',
'很多', '/', '领域', '/']
>>>
```

算法分析与评价：

该算法简单易懂，能解决 70% ~ 80%的常用词的分词问题，但性能较差。算法时间复杂度为 O(n×m)，其中 n 为待分词的中文字符串长度，由外层循环 while 控制，m 为最大分词长度，由内层循环 for 控制。最大词长的选择会影响每次循环判断的次数，遇到长的词语，如“中华人民共和国”，最大词长需要设置为 7，但是大多数常用词语一般词长为 2、3 或 4。当字典词条数目过多时，会大大增加每次循环比较的次数，严重影响算法的性能。此外，该算法没有解决分词的歧义问题。针对该算法有很多改进，如用正向最大匹配和逆向最大匹配相结合的方法，或者从哈希散列和前缀树等数据结构方向加以改进。

4. 类与继承

本例定义了三个类，分别为 Member 类、Student 类、Teacher 类，Student 类和 Teacher 类分别继承 Member 类，并且定义自己的 tell()方法，其中使用了 Member 类的方法。代码如下：

```
class Member:
    def   init   (self, name, age):
        self.name = name
        self.age = age
    def tell(self):
        print ('Name:%s,Age:%d' % (self.name, self.age))

class Student(Member):
```

```
    def  init  (self, name, age, marks):
        Member.  init  (self, name, age)
        self.marks = marks
    def tell(self):
        Member.tell(self)
        print ('Marks:%d' % self.marks)

class Teacher(Member):
    def  init  (self, name, age, salary):
        Member.  init  (self, name, age)
        self.salary = salary
    def tell(self):
        Member.tell(self)
        print ('Salary:%d' % self.salary)

Stu 1 = Student('Tom', 21, 77)
Stu 2 = Student('Tim', 19, 87)
Stu 3 = Student('Tam', 22, 93)
Tea 1 = Teacher('Mrs.Wang', 42, 5200)
Tea 2 = Teacher('Mr.Zhang', 39, 4800)
members = [Stu 1,Stu 2,Stu 3,Tea 1,Tea 2]
for mem in members:
    mem.tell()
```

执行以上代码，输出结果为：

```
Name:Tom,Age:21
Marks:77
Name:Tim,Age:19
Marks:87
Name:Tam,Age:22
Marks:93
Name:Mrs.Wang,Age:42
Salary:5200
Name:Mr.Zhang,Age:39
Salary:4800
>>>
```

5. 二元语法模型练习

(1)　问题。

输入：

①　语料库="把拆迁人民和企业的生产安排好，就是生活和生产要安排好。

20170512-06-008-027/m 企业/ns]nt[安排]保证[人民]..安排

要让企业有获得感，始终把人民放在心中最高位置。好人民保证好生活，人民生活为人民。"

②　测试句子="安排好人民生活"

输出：以“词”作为基元计算出现句子“安排好人民生活”的概率。

(2)　数学原理简述。

对于由 $W=W_1W_2\ldots W_n$ 构成的句子(其中 W_i 是词或者单个的字)，估计 P(W)的最直接的方法是利用词的 n-gram，即：

$$P(W)=P(W_1W_2W_3\cdots W_n)$$
$$=P(W_1)P(W_2\mid W_1)P(W_3\mid W_1W_2)\cdots P(W_n\mid W_1,W_2\cdots W_{n-1})$$

但是，利用词的 n-gram 直接估计 P(W) 的方法，在目前是不可行的，所以一般采用马尔可夫假设的估计方法，实现对 P(W) 的近似。马尔可夫假设任意一个词 W_i 出现的概率只与它前面的词 W_{i-1} 有关，于是把上面的公式简化成：

$$P(W)=P(W_1)P(W_2\mid W_1)P(W_3\mid W_2)...P(W_n\mid W_{n-1})$$

这里对应的统计语言模型是二元模型(bigram)，对应一阶马尔科夫链。如果一个词的出现仅依赖于它前面出现的两个词，那么就称为三元模型(trigram)，也可以假设一个词由前面 n−1 个词决定，对应的模型称为 n 元模型(n-gram)。

这里的关键问题是估算条件概率：

$$P(W_i\mid W_{i-1})=P(W_{i-1},W_i)/P(W_{i-1})$$

至此，问题转化为计算联合概率 $P(W_{i-1},W_i)$ 和边缘概率 $P(W_{i-1})$。

而计算联合概率 $P(W_{i-1},W_i)$ 和边缘概率 $P(W_{i-1})$ 时，只要计算语料库中 (W_{i-1},W_i) 这对词在统计的文本中前后相邻出现了多少次 $c(W_{i-1},W_i)$，以及 W_{i-1} 本身在同样的语料库中出现了多少次 $c(W_{i-1})$，然后用两个数分别除以语料库的大小 c，即可得到这些词或二元组的相对频度，再根据大数定理，只要统计量足够，相对频度就等于概率：

$$P(W_{i-1},W_i)\sim f(W_{i-1},W_i)=c(W_{i-1},W_i)/c；\ p(W_{i-1})\sim f(W_{i-1})=c(W_{i-1})/c$$

所以最后：

$$P(W_i|W_{i-1})=c(W_{i-1},W_i)/c(W_{i-1})$$

它们的积 P(W) 即可求出。

此处还要用到数据平滑处理技术，它是用来解决 0 概率的问题，产生更准确的概率来调整最大似然估计的一种技术，即提高低概率(如零概率)，降低高概率，尽量使概率分布趋于均匀。

对于二元文法模型来说，一种最简单的平滑技术就是假设每个二元文法出现的次数比实际出现的次数多一次，于是把该处理方法称为“加 1 法”，公式如下：

$$p(w_i\mid w_{i-1})=\frac{1+c(w_{i-1}w_i)}{\sum_{w_i}[1+c(w_{i-1}w_i)]}=\frac{1+c(w_{i-1}w_i)}{|C|+\sum_{w_i}c(w_{i-1}w_i)}$$

其中|C|为单词表中单词的个数，这里指分词后单词的个数，也就是语料库的大小。

(3) 各实现模块分析。

① 初始化函数：主要包括 6 个参数，两个列表、两个字典、两个数值变量。

分别为：保存语料库句子分词的列表、保存测试句子分词的列表、保存语料库句子分词的字典、保存测试句子分词的字典、概率变量、计数变量等。

② 格式化函数：利用正则表达式模块 re 的函数和列表 list 特性将句子内容转变为 BEGxxxENDBEGxxxEND 这种形式。如下面的例子。

语料库：BEG 把拆迁人民和企业的生产安排好，就是生活和生产要安排好 ENDBEG

20170512-06-008-027/m 企业/ns]nt[安排]保证[人民]..安排要让企业有获得感，始终把人民放在心中最高位置 ENDBEG 好人民保证好生活，人民生活为人民 END

测试句子：BEG 安排好人民生活 END

③　语料库句子中文分词并统计词频函数：使用 jieba 分词，首先用 suggest_freq (segment, True)调节单个词语的词频，使其能被分出来。然后分词并且统计词频，如果词在字典 self.dic_test{}中出现过则加 1，未出现则等于 1。如下面的例子。

分词结果：['BEG', '把', '拆迁', '人民', '和', '企业', '的', '生产', '安排', '好', '，', '就是', '生活', '和', '生产', '要', '安排', '好', 'END', 'BEG', '\n', '20170512', '-', '06', '-', '008', '-', '027', '，', 'm', '企业', '，', 'ns', ']', 'nt', '[', '安排', ']', '保证', '[', '人民', ']', '.', '.', '安排', '\n', '要', '让', '企业', '有', '获得', '感', '，', '始终', '把', '人民', '放在', '心中', '最高', '位置', 'END', 'BEG', '好', '人民', '保证', '好', '生活', '，', '人民', '生活', '为', '人民', 'END']

统计结果：{'': 4, '最高': 1, '始终': 1, '[': 2, ']': 3, '有': 1, '安排': 4, '20170512': 1, '拆迁': 1, '-': 3, '位置': 1, '\n': 2, '027': 1, '就是': 1, '.': 2, '和': 2, 'm': 1, '生活': 3, '感': 1, '008': 1, '人民': 6, '的': 1, '让': 1, '好': 4, 'nt': 1, 'BEG': 3, '生产': 2, 'END': 3, 'ns': 1, '心中': 1, '把': 2, '06': 1, '企业': 3, '为': 1, '，': 3, '要': 2, '保证': 2, '获得': 1, '放在': 1}

统计以后的字典是这样的，每次输出词典内容的顺序不固定。

④　测试句子中文分词并统计词频函数。

结果如下：

分词结果：['BEG', '安排', '好', '人民', '生活', 'END']

统计结果：{'人民': 1, 'END': 1, '好': 1, 'BEG': 1, '安排': 1, '生活': 1}

⑤　二元文法函数：计算两个紧邻词一起出现的个数，遍历语料字符串，如果测试和语料的两个紧邻的词都相同，则加 1。如：

['BEG', '安排', '好', '人民', '生活', 'END']

[0, 2, 1, 1, 0, 0]

结果意义：

“BEG 安排”在一起出现的次数为 0 次。

“安排好”在一起出现的次数为 2 次。

“好人民”在一起出现的次数为 1 次。

“人民生活”在一起出现的次数为 1 次。

“生活 END”在一起出现的次数为 0 次。

END 之后没有词，所以最后一个的次数为 0 次。

⑥　计算概率函数。

用数据平滑处理“加 1 法”，计算 $P(W_i \mid W_{i-1}) = P(W_{i-1}, W_i) / P(W_{i-1})$，并求它们的积。如：

0.25

0.15

0.06

0.017142857142857144

0.004285714285714285

0.0010714285714285713

输出最后结果：0.0010714285714285713

(4) Python 程序实现：

```
import jieba  #导入 jieba 分词模块
import re     #导入正则表达式模块

class GRAM2:
    """
    二元文法类:计算一句话在已知语料库中出现的概率
    类初始化
    字符串格式化
    分词并统计词频 cws1 和 cws2
    二元文法统计词出现的次数
    计算概率
    """
    list original = []      #保存语料库句子分词列表
    dic original = {}      #保存语料库句子分词字典
    list test = []         #保存测试句子分词的列表
    dic test = {}         #保存测试句子分词的字典
    list count = []       #保存语料库紧邻两个分词个数的列表
    p = 1               #概率变量初始化
    f = 0               #计数变量初始化
    def   init   (self,lo,do,lt,dt,lc,p,f):
        self.list original = lo
        self.dic original = do
        self.list test = lt
        self.dic test = dt
        self.list count = lc
        self.p = p
        self.f = f
    #格式化
    def form(self,string para):
        #存放正则表达式处理过的字符串
        tmp = []
        for i in range(len(string para)):
            #去除类似这样的词"20170512-06-008-027/m"
            if re.compile(r'\d+\-\S+').match(string para[i]):
                continue
            else:
                #将类似的词“中国梦/ns”替换为“中国梦”
                data = re.compile(r'\/\w+').sub('\n',string para[i])
                #找到以“[”或“]”开头的词
                if re.compile(r'(\[\S+)|(\]\S+)').match(data):
                    #去除“]nt[北京”,“]天津”,“[河北”,“]..中国梦”
                     #三类词的头部无用部分(先匹配长的部分)
                    ok data =
                      re.compile(r'(\]\w+\[)|(\])|(\[)|(\]..)').sub('',data)
                    tmp.append(ok data)
                    continue
                tmp.append(data)
        #将 tmp 列表转换为字符串
```

```
        tmp = "".join(tmp)
        #如果是以“。”结束的则将“。”删掉，删掉最后一个。不包括句子中的“。”
        if tmp.endswith("。"):
            tmp = tmp[:-1]
        #添加起始符 BEG 和终止符 END，并替换句子中出现的“。”
        #将句子变为“BEGxxxENDxxxxBEGxxxEND”这种形式
        string1 = "BEG" + tmp.replace("。", "ENDBEG") + "END"
        return string1
    #中文分词并统计词频--语料库
    def cws1(self,string para):
        #使用 suggest freq(segment, True) 调节单个词语的词频，使其能被分出来。
        jieba.suggest freq("BEG", True)
        jieba.suggest freq("END", True)
        #jieba 分词
        string para = jieba.cut(string para,HMM = False)
        string para form = "/".join(string para)
        #将词按“/”分割后依次填入列表 lists
        self.list original = string para form.split("/")
        #统计字符串词频，如果词在字典 ori dict{}中出现过则+1，未出现则=1。
        for word in self.list original:
            self.dic original[word] = (self.dic original[word] + 1
if word in self.dic original else 1)
        print("----------语料库句子中文分词结果为：")
        print (self.list original)
        print("----------分词并统计词频的结果为：")
        print(self.dic original)
        return self.list original

    #中文分词并统计词频—测试句子
    def cws2(self,string para):
        #使用 suggest freq(segment, True) 调节单个词语的词频，使其能被分出来
        jieba.suggest freq("BEG", True)
        jieba.suggest freq("END", True)
        #jieba 分词
        string para = jieba.cut(string para,HMM=False)
        string para form = "/".join(string para)
        #将词按“/”分割后依次填入 lists
        self.list test = string para form.split("/")
        #统计词频，如果词在字典 dicts{}中出现过则+1，未出现则=1
        for word in self.list test:
            self.dic test[word] =
             (self.dic test[word] + 1 if word in self.dic test else 1)
        print("----------测试句子中文分词结果为：")
        print (self.list test)
        print("----------分词并统计词频的结果为：")
        print(self.dic test)
        return self.list test

    #二元文法计算两个紧邻词一起出现的个数
    def bigram(self,list_para1,list_para2):
```

```
        #list count 赋值
        initial value = 0
        list length = len(list para2)
        self.list count = [initial value] * (list length)
        #遍历测试的字符串
        for i in range(len(list para2)-1):
            #遍历语料字符串，且因为是二元文法
            #不用比较语料字符串的最后一个字符
            for j in range(len(list para1)-2):
                #如果测试和语料的二个紧邻的词都相同，则加 1
                if list para2[i]==list para1[j]
                  and list para2[i+1] == list para1[j+1]:
                    self.list count[i] += 1
        print("----------二元文法函数结果为：")
        print(self.list count)
        return

    #计算概率 P(Wi|Wi-1)=P(Wi-1,Wi)/P(Wi-1)，并求它们的积
    def probability(self,list para):
        print("----------概率函数中间过程为：")
        for word in list para:
            #数据平滑处理：加 1 法
            self.p=self.p*(float(self.list count[self.f]+1)/
               float(self.dic original[word]+1))
            self.f = self.f + 1
            #print("概率函数中间过程为：")
            print(self.p)
        print("----------概率函数结果为：")
        print(self.p)
        return

if   name   == "  main  ":
    #测试函数
    def test():
        string1 = """把拆迁人民和企业的生产安排好，就是生活和生产要安排好。
要让企业有获得感，始终把人民放在心中最高位置。好人民保证好生活，人民生活为人民。"""
        string2 = "安排好人民生活"
        list1 = []
        list2 = []
        gram2 = GRAM2([],{},[],{},[],1,0)
        list1 = gram2.cws1(gram2.form(string1))
        list2 = gram2.cws2(gram2.form(string2))
        gram2.bigram(list1,list2)
        gram2.probability(list2)
        return
    #调用测试函数
    test()
```

输出结果如下：

```
Building prefix dict from the default dictionary ...
Loading model from cache C:\Users\ADMINI~1\AppData\Local\Temp\jieba.cache
Loading model cost 1.103 seconds.
Prefix dict has been built succesfully.
----------语料库句子中文分词结果为:
['BEG', '把', '拆迁', '人民', '和', '企业', '的', '生产', '安排', '好', '，', '就是', '生活', '和', '生产', '要', '安排', '好', 'END', 'BEG', '\n', '要', '让', '企业', '有', '获得', '感', '，', '始终', '把', '人民', '放在', '心中', '最高', '位置', 'END', 'BEG', '好', '人民', '保证', '好', '生活', '，', '人民', '生活', '为', '人民', 'END']
----------分词并统计词频的结果为:
{'BEG': 3, '生产': 2, '好': 4, '安排': 2, '的': 1, '和': 2, '有': 1, '人民': 5, '感': 1, '保证': 1, '位置': 1, '企业': 2, '放在': 1, '就是': 1, '生活': 3, '获得': 1, '最高': 1, 'END': 3, '\n': 1, '让': 1, '拆迁': 1, '心中': 1, '要': 2, '为': 1, '把': 2, '始终': 1, '，': 3}
----------测试句子中文分词结果为:
['BEG', '安排', '好', '人民', '生活', 'END']
----------二元文法函数结果为:
[0, 2, 1, 1, 0, 0]
----------概率函数中间过程为:
0.25
0.25
0.1
0.03333333333333333
0.008333333333333333
0.0020833333333333333
----------概率函数结果为:
0.0020833333333333333
>>>
```

6. 维特比算法练习

维特比算法是解决隐马尔可夫模型预测问题的算法之一，本例是对李航的《统计学习方法》一书中的第 10 章例题 10.3 的 Python 实现，详细原理和算法过程见参考文献[17]。

Python 代码如下：

```
class HMM:
    """隐马尔可夫模型类"""
    states = ()       #隐状态
    obs = ()         #观测序列
    start p = {}      #初始概率(隐状态)
    trans p = {}     #转移概率(隐状态)
    emit p = {}    #发射概率(隐状态表现为显状态的概率)
    V = [{}]        #路径概率表 V[时间][隐状态] = 概率，Viterbi 函数中使用
    def   init  (self,s,o,sp,tp,ep,v):
        self.states = s
        self.obs = o
        self.start p = sp
        self.trans p = tp
        self.emit p = ep
        self.V = v

    #打印路径概率表
    def print list(self):
        for j in self.V[0].keys():
            print('{0:5s}'.format(j))
            for t in range(0,len(self.V)):
```

```
                print ('{0:6f}'.format(self.V[t][j]))
            print()

    def Viterbi(self):
        #Viterbi 算法
        #一个中间变量，代表当前状态是哪个隐状态
        path = {}
        #初始化初始状态 (t == 0)
        for y in self.states:
            self.V[0][y] = self.start p[y] * self.emit p[y][self.obs[0]]
            path[y] = [y]
        #对 t > 0 跑一遍维特比算法
        for t in range(1,len(self.obs)):
            self.V.append({})
            newpath = {}
            for y in self.states:
                #概率，隐状态=前状态是 y0 的概率 * y0 转移到 y 的概率*
#y 表现为当前状态的概率
   (prob,state)=max([(self.V[t-1][y0]*self.trans p[y0][y]*
self.emit p[y][self.obs[t]], y0) for y0 in self.states])
                #记录最大概率
                self.V[t][y] = prob
                #记录路径
                newpath[y] = path[state] + [y]
            #不需要保留旧路径
            path = newpath
        (prob, state) = max([(self.V[len(self.obs) - 1][y], y) for y in
self.states])
        return (prob, path[state])

if   name  ==" main  ":
    states = ('box1', 'box2', 'box3')
    observations = ('red', 'white', 'red')
    start probability = {'box1': 0.2, 'box2': 0.4, 'box3': 0.4}
    transition probability = {
        'box1' : {'box1': 0.5, 'box2': 0.2, 'box3': 0.3},
        'box2' : {'box1': 0.3, 'box2': 0.5, 'box3': 0.2},
        'box3' : {'box1': 0.2, 'box2': 0.3, 'box3': 0.5},
    }
    emission probability = {
        'box1' : {'red': 0.5, 'white': 0.5},
        'box2' : {'red': 0.4, 'white': 0.6},
        'box3' : {'red': 0.7, 'white': 0.3},
    }
    hmm = HMM(states,
              observations,
              start probability,
              transition probability,
              emission probability,
              [{}])
```

```
    print (hmm.Viterbi())
    print (hmm.print_list())
```

输出结果如下图 6-6 所示。

```
>>> ============================== RESTART ==============================
>>>
(0.014699999999999998, ['box3', 'box3', 'box3'])
box2
0.160000
0.050400
0.010080

box3
0.280000
0.042000
0.014700

box1
0.100000
0.028000
0.007560

None
>>>
```

图 6-6　输出结果

本 章 小 结

本章对面向对象技术进行了全面介绍。首先介绍了类的概念、类的定义与使用，Python 中类的作用域与使用方法，并举例说明；其次介绍了类对象支持的两种操作，介绍了类的属性和实例属性及命名约定，并对 self 的使用情况做了说明；然后介绍了单继承和多继承的概念，介绍了方法重写等，并对 isinstance()函数、super()函数等分别举例说明；最后的案例实训列举了六个典型案例：堆、栈、中文分词、类的继承、二元语法模型、维特比算法。

习　　题

1. 填空题

(1)　在 Python 中，定义类的关键字是(　　　)。

(2)　类的定义如下：

```
class person:
    name = 'zhangsan'
    score = 89
```

该类的类名是(　　　)，其中定义了(　　　)属性和(　　　)属性，它们都是(　　　)属性。如果在属性名前加两个下划线(__)，则该属性是(　　　)属性。将该类实例化，创建对象 p，使用的语句为(　　　)，通过 p 来访问属性，格式为(　　　)、(　　　)。

(3) 可以从现有的类来定义新的类，这称为类的(　　　)，新的类称为(　　　)，而原来的类称为(　　　)、父类或超类。

(4) 创建对象后，可以使用(　　　)运算符来调用其成员。

(5) 下列程序的运行结果为(　　　):

```
class  test:
   def   init  (self,id):
      self.id=id
      id=345
t=test(123)
print (t.id)
```

(6) 下列程序的运行结果为(　　　):

```
class teacher:
   def   init  (self,par):
      self.a=par
class student (teacher):
   def   init  (self,par):
      teacher.  init  (self,par)
      self.b=par
stu=student(30)
print (stu.a,stu.b)
```

2. 选择题

(1) 下列说法中不正确的是(　　)。

A. 类是对象的模板，而对象是类的实例

B. 实例属性姓名如果以__(双下划线)开头，就变成了一个私有变量

C. 只有在类的内部才可以访问类的私有变量，在外部不能访问

D. 在 Python 中，一个子类只能有一个父类

(2) 下列选项中不是面向对象程序设计基本特征的是(　　)。

A. 继承　　B. 多态　　C. 可维护性　　D. 封装

(3) 在方法定义中，访问实例属性 x 的格式是(　　)。

A. x　　B. self.x　　C. self[x]　　D. self.getx( )

(4) 下列程序的执行结果是(　　)。

```
class Point:
   x=15
   y=5
   def   init  (self,x,y):
      self.y=y
p=Point(25,25)
print(p.x,p.y)
```

A. 15 25　　B. 25 15　　C. 5 15　　D. 25 25

(5) 下列程序的执行结果是(　　)。

```
class A():
    a=15
class A1(A):
    pass
print (A.a,A1.a)
```

A. 15 15　　B. 15 pass　　C. pass 15　　D. 运行出错

3. 问答题

(1) 面向对象程序设计的三要素是什么?

(2) 简单解释 Python 中以下划线开头的变量名之特点。

(3) 在 Python 中导入模块中的对象有哪几种方式?

(4) Python 生成一个随机数的方法有哪些?

4. 实验操作题

(1) 定义动物类，跑为属性，再定义一个猫类，名字为属性，实现继承功能，并实例化调用。

(2) 定义 Person 类，生成 Student 类，填写新的函数用来设置学生专业，然后生成该类对象并显示信息。

(3) 设计一个三维向量类，并且实现向量的加法、减法以及向量与标量的乘法和除法运算。

(4) 编写类并且定义函数，可以接收任意多个整数，并输出其中的最大值和所有整数之和。

(5) 定义类并编写函数，可以接收一个字符串，分别统计大写字母、小写字母、数字、其他字符的个数，并以元组的形式返回结果。



第 7 章

数据库编程

本章要点

(1) 数据库技术基础。
(2) SQLite 数据库的数据类型、基本操作。
(3) MySQL 数据库的数据类型、基本操作。
(4) 使用 Python 操作 SQLite 数据库、MySQL 数据库。

学习目标

(1) 了解数据库的基本概念。
(2) 理解 SQLite 数据库和 MySQL 数据库的数据类型。
(3) 熟悉 SQLite 数据库和 MySQL 数据库的基本操作。
(4) 掌握如何使用 Python 操作 SQLite 数据库与 MySQL 数据库。

前面各章对 Python 语言的基础知识进行了详细的介绍，本章及后面各章将陆续介绍 Python 的一些实际应用，旨在让读者能更加清晰地了解 Python 这门编程语言的魅力。本章主要介绍如何使用 Python 进行数据库开发。

一般情况下，绝大多数应用程序都需要使用数据库来存放数据。Python 支持多种数据库，使用相应的模块即可连接到数据库进行编程。

因本书篇幅所限，本章仅介绍两种关系型数据库——SQLite 数据库与 MySQL 数据库，重点介绍在 Python 中如何进行数据库编程。

7.1 数据库技术基础

7.1.1 数据库的基本概念

数据库(Database，DB)是存储数据的仓库，是长期存放在计算机内、有组织、可共享的大量数据的集合。数据库中的数据按照一定的数据模型组织、描述和存储，具有尽可能小的冗余度，同时，具有较高的独立性和易扩展性。

数据库管理系统(Database Management System，DBMS)是位于用户与操作系统之间的一层数据库管理软件，帮助用户向计算机中输入、管理大量的数据，方便用户定义数据、操作数据和维护数据。其主要功能包括如下几种。

(1) 数据定义功能：提供数据定义语言(Data Definition Language，DDL)，方便用户对数据库中的数据对象进行定义。

(2) 数据操作功能：提供数据操纵语言(Data Manipulation Language，DML)，可以对数据库中的数据进行查询、插入、删除和修改等基本操作。

(3) 数据库的管理和维护：对数据库进行统一管理控制，以保证其安全性、完整性、一致性，并实现多用户环境下的并发使用。

数据库系统(Database System，DBS)是指在计算机系统中引入数据库后组成的系统。数据库系统一般由数据库、操作系统、数据库管理系统(及其开发工具)、应用系统、数据库管理员和用户组成，如图 7-1 所示。

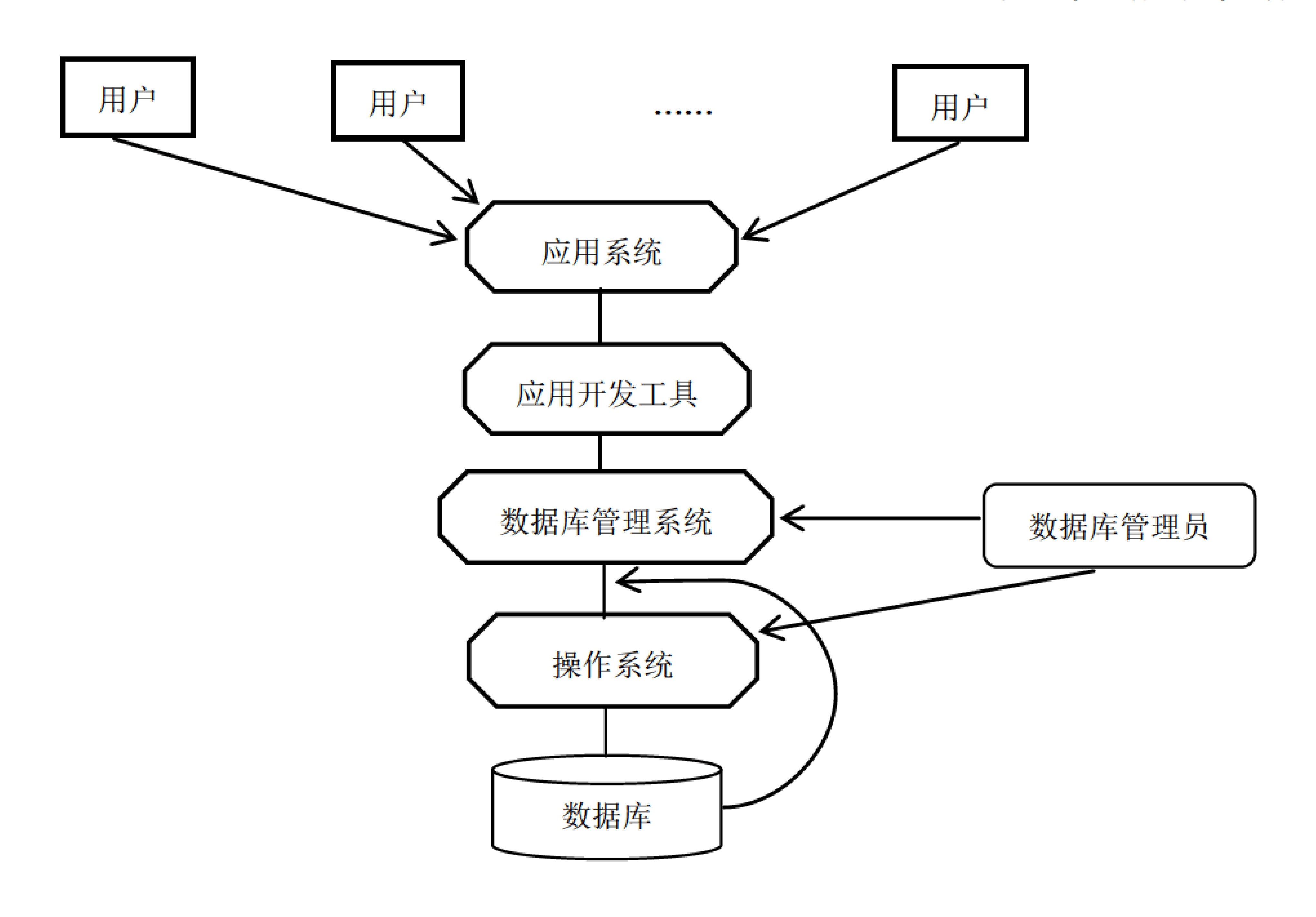


图 7-1　数据库系统的组成

数据库、数据库管理系统和数据库系统是三个不同的概念，数据库强调的是数据，是数据库管理系统的管理对象；数据库管理系统强调的是管理软件，是数据库系统的组成部分，而数据库系统强调的是系统。

7.1.2　数据库的类型

根据数据存储模型，可将数据库分为层次数据库、网状数据库、关系数据库、面向对象数据库等。目前常见的数据库主要有两大类，即关系型数据库和非关系型数据库，本节主要介绍前者。目前常见的关系型数据库有 Oracle、MySQL、SQL Server、SQLite、DB2、Sybase、Informix 等。

关系型数据库是建立在关系模型基础上的。关系模型是指用二维表的形式来表示实体和实体间联系的数据模型。实体是指现实世界中具有一定特征或属性并客观存在的数据对象，实体与实体间的联系可以分为以下三种。

(1)　一对一联系：如一个工厂只能有一个厂长，而一个厂长只能在一个工厂任职，工厂和厂长为一对一的联系。

(2)　一对多联系：如一个班级有多名学生，而一名学生只能在一个班级里，班级和学生为一对多的联系。

(3)　多对多联系：如一名学生可以选择多门选修课程，而一门选修课程可以被多名学生选择，学生和课程是多对多的联系。

在关系模型中，一个关系对应着一张二维表，一张二维表由行和列组成。表的一行称为一个记录，描述一个具体实体的一组数据，例如表 7-1 所示的学生信息表中，学号为

201701、姓名为李丽、性别为女、出生日期为 1993-01-12、籍贯为天津的一组数据为一个记录。表的列称为字段，描述实体的一个特征或属性，例如表 7-1 所示的学生信息表中的学号、姓名、性别、出生日期、籍贯就是字段。每个表中通常都有一个关键字，一个可以唯一标识一条记录的字段，例如表 7-1 中的学号。

表 7-1　学生信息表

学号	姓名	性别	出生日期	籍贯
201701	李丽	女	1993-01-12	天津
201702	王阳	男	1993-02-22	唐山
201703	张亮	男	1994-12-01	北京
……	……	……	……	……

关系数据库最大的特点是事务的一致性，这使得它并不适用于大数据量的 Web 系统，于是，非关系型数据库应运而生。NoSQL(Not Only SQL)泛指非关系型的数据库，NoSQL 数据库基本上不进行复杂的处理，只应用在特定的领域，是对传统关系型数据库的一个有效补充。由于它可以为大数据建立快速、可扩展的存储库，因而得到了非常迅速的发展。

非关系型数据库可以分为四大类：键值对存储(key-value store)数据库，如 Redis、Voldemort、Tokyo Cabinet/Tyrant、Oracle BDB 等；列存储(Column-oriented)数据库，如 Cassandra、HBase、Riak 等；文档存储(Document Store)数据库，如 CouchDB、MongoDb 等；图形(Graph)数据库，例如 Neo4J、InfoGrid、Infinite Graph 等。在这四类非关系型数据库中，每一种都可解决相应的问题，这些问题是关系型数据库所不能解决的。

7.2　SQLite 数据库

SQLite 数据库是一款非常小巧的开源嵌入式数据库，占用的资源非常低，能够支持 Windows、Linux、Unix 等主流操作系统，同时，能跟很多编程语言结合，比如 Python、C#、PHP、Java 等。

7.2.1　SQLite 数据库的下载和安装

进入 SQLite 下载页面：http://www.sqlite.org/download.html，在如图 7-2 所示的界面找到 Precompiled Binaries for Windows 一项，下载 Windows 下的预编译二进制文件包：sqlite-tools-win32-x86-<build#>.zip 和 sqlite-dll-win32-x86-<build#>.zip(<build#>是 sqlite 的编译版本号)。

当前最新版本的安装包为 sqlite-tools-win32-x86-3180000.zip，对应的版本是 sqlite3。将安装包下载并解压到磁盘中，并将解压后的目录添加到系统的 PATH 环境变量中(加入 PATH 环境变量是为了直接在命令行上使用 sqlite3)。

打开 DOS 命令提示符窗口，输入 sqlite3 命令并按 Enter 键，如果安装成功，则显示如图 7-3 所示的信息。

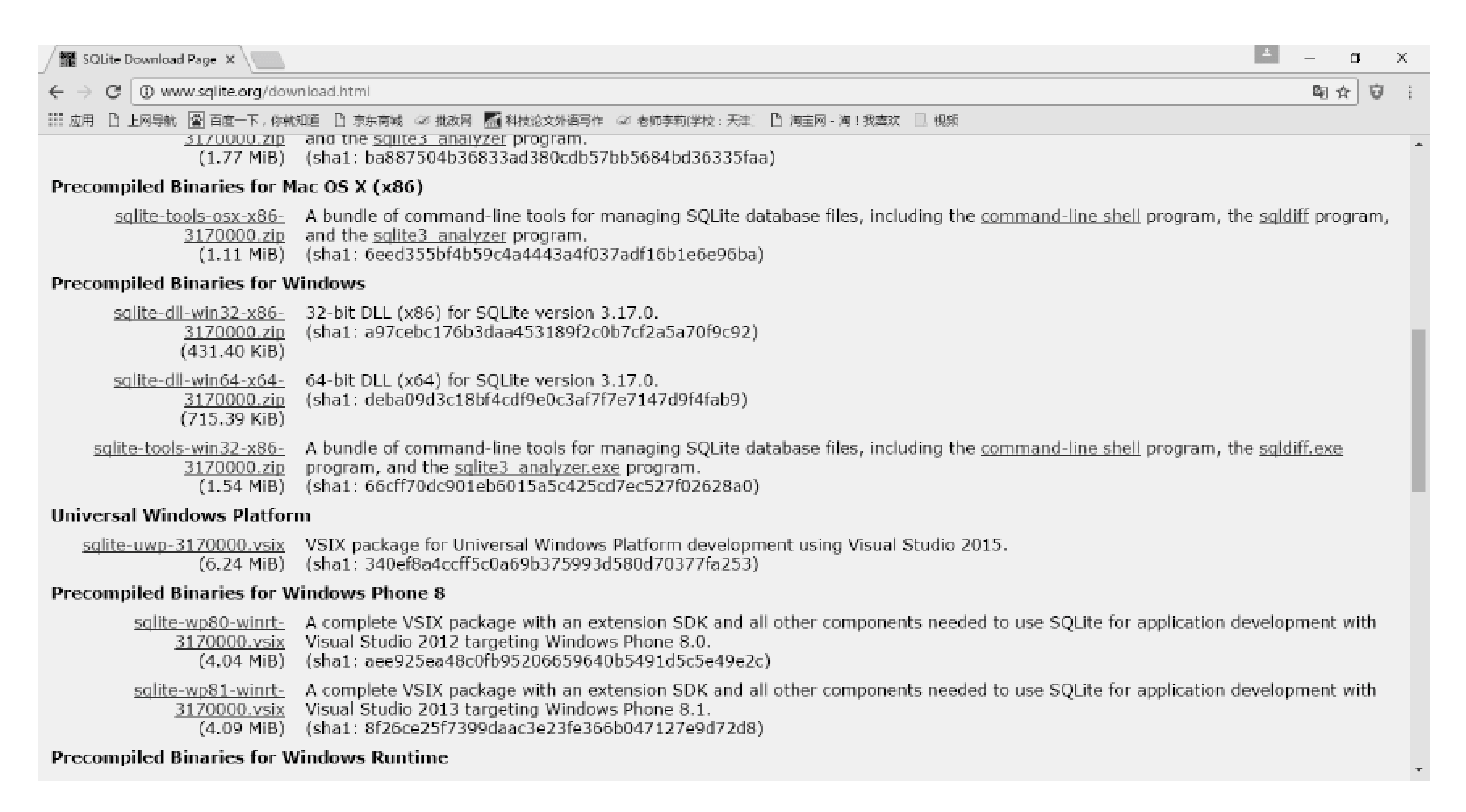


图 7-2　SQLite 下载页界面



图 7-3　SQLite 命令行窗口

7.2.2　SQLite 数据类型

SQLite 支持的数据类型如表 7-2 所示。

表 7-2　SQLite 支持的数据类型

数据类型	描　述
NULL	值是一个空值
INTEGER	值是一个带符号的整数
REAL	值是一个浮点数，存储为 8 字节的 IEEE 浮点数
TEXT	值是一个文本字符串，使用数据库编码存储
BLOB	值是一个二进制数，完全根据它的输入存储

但实际上，SQLite3 也接受如表 7-3 所示的数据类型。

表 7-3　SQLite3 支持的数据类型

数据类型	描　述
SMALLINT	16 位的整数
DECIMAL(P,S)	小数，P 指数字的位数，S 指小数点后数的位数，如果没有特别指定，则系统会默认为 P=5，S=0

续表

数据类型	描　述
FLOAT	32 位的浮点数
DOUBLE	64 位的实数
CHAR(N)	长度为 N 的字符串，N≤254
VARCHAR(N)	可变长度且其最大长度为 N 的字符串，N≤4000
GRAPHIC(N)	和 CHAR(N)一样，不过其单位是两个字节，N≤127。这个形态是为了支持两个字节长度的字体，例如中文文字
VARGRAPHIC(N)	可变长度且其最大长度为 N 的双字节字符串，N≤4000
DATE	包含了年、月、日
TIME	包含了时、分、秒
TIMESTAMP	包含了年、月、日、时、分、秒、千分之一秒
DATETIME	包含日期和时间

7.2.3　创建 SQLite 数据库

在运行 SQLite 数据库的同时，可通过下面的命令创建数据库：

```
sqlite3 数据库文件名
```

SQLite 数据库文件的扩展名为.db。如果指定的数据库文件存在，则打开数据库；否则在当前目录下创建该数据库，并通过下面的命令来保存数据库：

```
.save 数据库文件名
```

【例 7-1】创建 SQLite 数据库 SQLitedb.db：

```
sqlite3 SQLitedb.db
```

注意： 该命令是在 DOS 命令提示符下输入的，并不是在 SQLite 命令提示符下输入的。

7.2.4　SQLite 的基本操作

1. 创建表

使用 CREATE TABLE 语句创建表的语法格式为：

```
CREATE TABLE 表名
(
      列名 1 数据类型 字段属性,
      列名 2 数据类型 字段属性,
      ...
      列名 n 数据类型 字段属性
);
```

常用的字段属性如表 7-4 所示。

表 7-4　常用的字段属性

字段属性	描　述
PRIMARY KEY	设置指定列为主键，用于确保记录的唯一性
NOT NULL	设置指定列的值不允许为空
UNIQUE	设置指定列所有值除 NULL 外都不相同
DEFAULT	设置指定列的默认值
CHECK	设置指定列的检查条件，确保指定列中的所有值满足该条件

【例 7-2】创建学生课程表 Course，表结构如表 7-5 所示。

表 7-5　表 Course 的结构

字　段	数据类型	字段说明
CNo	char(4)	课程编号(主键)
CName	varchar(50)	课程名称(不为空)
CCredits	decimal(4,1)	学分(默认为 4)
CTime	decimal(3,0)	总学时
CTerm	char(11)	学期

使用 CREATE TABLE 语句创建表 Course 的语句为：

```
CREATE TABLE Course
   (
        CNo char(4) PRIMARY KEY,
        CName varchar(50) NOT NULL,
        CCredits decimal(4,1) DEFAULT(4),
        CTime decimal(3,0),
        CTerm char(11)
   );
```

执行下面的语句可以查看当前数据库中所有的表：

```
.tables
```

使用下列语句可以查看表的结构：

```
select * from sqlite_master where type='table' and name='表名';
```

或者：

```
.schema
```

2. 向表中插入数据

一旦把表创建好，就可以向表内插入数据了。可以使用 INSERT 语句向表内插入数据，语法格式为(列与值必须一一对应)：

```
INSERT INTO 表名(列名 1, 列名 2, ..., 列名 n)
VALUES(值 1, 值 2, ..., 值 n);
```

【例 7-3】参照表 7-6，向 Course 表中插入数据。

表 7-6　表 Course 中插入的数据

CNo	CName	CCredits	CTime	CTerm
0001	英语	2	36	2
0002	高等数学	3	36	2
0003	数据结构	4	54	2
0004	c 语言	3	54	2
0005	数据库系统概论	2	18	2
0006	操作系统	2	18	2

INSERT 语句如下：

```
INSERT INTO Course(CNo,CName,CCredits,CTime,CTerm) VALUES ('0001',
'英语',2,36,'2');
INSERT INTO Course(CNo,CName,CCredits,CTime,CTerm) VALUES ('0002',
'高等数学',3,36,'2');
INSERT INTO Course(CNo,CName,CTime,CTerm) VALUES ('0003','数据结构',
54,'2');
INSERT INTO Course(CNo,CName,CCredits,CTime,CTerm) VALUES ('0004',
'c 语言',3,54,'2');
INSERT INTO Course(CNo,CName,CCredits,CTime,CTerm) VALUES ('0005',
'数据库系统概论',2,18,'2');
INSERT INTO Course(CNo,CName,CCredits,CTime,CTerm) VALUES ('0006',
'操作系统',2,18,'2');
```

使用可视化工具 SQLiteStudio 可显示例 7-3 所创建的表 Course，如图 7-4 所示。



	CNo	CName	CCredits	CTime	CTerm
1	0001	英语	2	36	2
2	0002	高等数学	3	36	2
3	0003	数据结构	4	54	2
4	0004	c语言	3	54	2
5	0005	数据库系统概论	2	18	2
6	0006	操作系统	2	18	2

图 7-4　例 7-3 所创建的表 Course

注意：为了方便读者查询各条语句执行后的结果，部分例题的结果显示使用了可视化工具 SQLiteStudio。

3. 修改表中的数据

修改数据可以使用 UPDATE 语句来实现，其语法格式为：

```
UPDATE 表名 SET 列名 1=值 1, 列名 2=值 2, ..., 列名 n=值 n
WHERE 条件表达式;
```

该语句的功能是修改表中满足 WHERE 子句条件的记录。其中 SET 子句用于指定修改方法，即列 1 的值被设置为值 1，列 2 的值被设置为值 2，列 n 的值被设置为值 n。如果省略 WHERE 子句，则表中所有的记录都将被修改。

【例 7-4】将表 Course 中 CName 为操作系统的记录中的学时 CTime 修改为 36：

```
UPDATE Course SET CTime=36 WHERE CName='操作系统';
```

4. 删除数据

随着使用和对数据的修改，表中可能存在一些无用的数据。可以使用 DELETE 语句删除表中的数据，其语法格式如下：

```
DELETE FROM 表名 WHERE 删除条件;
```

DELETE 语句的功能是将指定表中满足 WHERE 子句条件的所有记录删除。如果没有提供 WHERE 子句，则 DELETE 语句将删除表中的所有记录，但表的结构仍在，也就是说，DELETE 语句删除的只是表中的数据。

【例 7-5】删除表 Course 中课程号 CNo 为 0004 的记录：

```
DELETE FROM Course WHERE CNo='0004';
```

5. 查询数据

使用 SELECT 语句查询表中的数据，语句的一般格式为：

```
SELECT 列名 1, 列名 2, ..., 列名 n FROM 表名 WHERE 查询条件;
```

当执行 SELECT 语句时，指定的表中所有满足 WHERE 子句条件的数据都将被返回。如果没有提供 WHERE 子句，则 SELECT 语句将返回所有记录中指定的字段值。如果要查询表中的所有字段，可以用“*”代替“列名 1, 列名 2, ..., 列名 n”。

【例 7-6】查询表 Course 中学分 CCredits 为 2 的课程编号 CNo 和课程名称 CName：

```
SELECT CNo,CName FROM Course WHERE CCredits=2;
```

查询结果如图 7-5 所示。

【例 7-7】返回表 Course 中的所有信息：

```
SELECT * FROM Course;
```

查询结果如图 7-6 所示。

	CNo	CName
1	0001	英语
2	0005	数据库系统概论
3	0006	操作系统

图 7-5　例 7-6 的查询结果

	CNo	CName	CCredits	CTime	CTerm
1	0001	英语	2	36	2
2	0002	高等数学	3	36	2
3	0003	数据结构	4	54	2
4	0005	数据库系统概论	2	18	2
5	0006	操作系统	2	36	2

图 7-6　例 7-7 的查询结果

注意： 例 7-7 的查询是在例 7-4 和例 7-5 的基础上，故在查询结果中 CName 为操作系统的 CTime，已更改为 36，而且删除了 CNo 为 0004 的记录。

7.2.5　使用 Python 操作 SQLite 数据库

Python 标准库中带有 sqlite3 模块，使用 sqlite3 模块操作数据库的基本步骤如下。

1. 导入 sqlite3 模块

使用如下语句从 Python 标准库导入 sqlite3 模块：

```
import sqlite3
```

2. 建立数据库连接

调用数据库模块中的 connect()方法建立数据库连接，指定数据库文件名。语法格式如下：

```
数据库连接对象=sqlite3.connect(数据库名)
```

【例 7-8】使用 connect()方法在本地磁盘 D 中创建数据库 test.db：

```
co=sqlite3.connect(r"D:\test.db")
```

数据库名是包含绝对路径的数据库文件名。如果 D:\test.db 存在，则打开数据库；否则创建并打开数据库 D:\test.db。打开数据库时返回的对象 co 就是一个数据库连接对象。

使用如下方法可以创建一个内存数据库：

```
数据库连接对象=sqlite3.connect(":memory:")
```

3. 创建游标对象

调用 cursor()方法创建游标对象：

```
游标对象=数据库连接对象.cursor()
```

4. 调用 execute()方法执行 SQL 语句

调用 execute()方法执行 SQL 语句的具体方法如表 7-7 所示。

表 7-7　调用 execute()方法执行 SQL 语句的具体方法

具体方法	描　述
游标对象.execute(sql)	执行一条 SQL 语句
游标对象.execute(sql, parameters)	执行一条带参数的 SQL 语句
游标对象.executemany(sql, parameters)	执行多条带参数的 SQL 语句
游标对象.executescript(sql_script)	执行 SQL 脚本

SQL 语句中的参数可以使用占位符“?”代替，并在随后的传递参数中使用元组给出具体值，或使用命名参数，传递参数使用字典。

【例 7-9】在数据库 D:\test.db 中使用 execute()方法执行 SQL 语句创建表 Course，并插入数据：

```
import sqlite3
co=sqlite3.connect(r"D:\test.db")
cu=co.cursor()
```

```
cu.execute("CREATE TABLE Course(CNo char(4) PRIMARY KEY,CName varchar(50)
not  null,CCredits decimal(4,1) default(4),CTime decimal(3,0),
CTerm char(11))")
cu.execute("INSERT INTO Course(CNo,CName,CTime,CTerm) VALUES ('0003',
'数据结构',54,'2')")
cu.execute("INSERT INTO Course VALUES (?,?,?,?,?)",('0001',
'英语',2,36,'2'))
a=[('0002','高等数学',3,36,'2'),('0004','c语言',3,54,'2'),('0005',
'数据库系统概论',2,18,'2'),('0006','操作系统',2,18,'2')]
cu.executemany("INSERT INTO Course VALUES (?,?,?,?,?)",a)
co.commit()  #提交数据
```

使用可视化工具 SQLiteStudio 可查看到例 7-9 所创建的表 Course，如图 7-7 所示。

	CNo	CName	CCredits	CTime	CTerm
1	0003	数据结构	4	54	2
2	0001	英语	2	36	2
3	0002	高等数学	3	36	2
4	0004	c语言	3	54	2
5	0005	数据库系统概论	2	18	2
6	0006	操作系统	2	18	2

图 7-7　例 7-9 所创建的表 Course

5. 获取游标的查询结果

获取游标的查询结果的具体方法如表 7-8 所示。

表 7-8　获取游标的查询结果的具体方法

具体方法	描　述
游标对象.fetchone()	获取结果集的下一条记录，无数据时返回 None
游标对象.fetchmany(n)	获取结果集中的 n 条记录，无数据时返回空 list
游标对象.fetchall()	获取结果集中的所有记录，无数据时返回空 list
游标对象.rowcount()	获取影响的行数、结果集的行数

【例 7-10】在数据库 D:\test.db 中使用游标查询表 Course 中的数据：

```
import sqlite3
co=sqlite3.connect(r"D:\test.db")
cu=co.cursor()
cu.execute("SELECT * FROM Course ORDER BY CNo")
print(cu.fetchone())
print(cu.fetchall())
```

查询结果如图 7-8 所示。

```
<sqlite3.Cursor object at 0x000002ACE9199180>
>>> print(cu.fetchone())
('0001', '英语', 2, 36, '2')
>>> print(cu.fetchall())
[('0002', '高等数学', 3, 36, '2'), ('0003', '数据结构', 4, 54, '2'), ('0004',
'c语言', 3, 54, '2'), ('0005', '数据库系统概论', 2, 18, '2'), ('0006', '操作系
统', 2, 18, '2')]
>>> print(cu.fetchone())
None
>>> |
```

图 7-8　例 7-10 的查询结果

6. 数据库的提交和回滚

(1) 提交数据库。

语法格式：数据库连接对象.commit()

功能：提交当前事务。

注意： 如果关闭数据库连接前未调用 commit()方法，则自上一次调用 commit()方法以来对数据库的更改全部丢失。

(2) 回滚数据库。

语法格式：数据库连接对象.rollback()

功能：回滚自上一次调用 commit()方法后对数据库所做的更改。

7. 关闭 cursor 对象和 connect 对象

(1) 关闭游标对象：

```
游标对象.close()
```

(2) 关闭数据库连接对象：

```
数据库连接对象.close()
```

数据库连接对象使用完毕后，应养成及时关闭的好习惯，以免造成数据丢失。

7.3 MySQL 数据库

MySQL 是一个多用户、多线程的关系型数据库管理系统，其工作模式是基于客户机/服务器结构的，具有开放性、多线程、支持多种 API、跨数据库连接、国际化、巨大的数据库规模等特点。目前它可以支持几乎所有的操作系统，包括 Windows 系列以及 Unix 系列等操作系统。由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，许多中小型网站为了降低网站总体成本，而选择 MySQL 作为网站数据库。

7.3.1 MySQL 数据库的下载和安装

访问 MySQL 网址 http://dev.mysql.com/downloads/，下载 mysql-installer-community-5.7.17.0.msi 文件。本节以 MySQL5.7.17 为例，介绍 MySQL 数据库的安装过程。

双击 mysql-installer-community-5.7.17.0.msi 文件，打开 MySQL Installer 安装向导，出现如图 7-9 所示的界面，选中 I accept the license terms 复选框，然后单击 Next 按钮，进入配置安装类型界面，如图 7-10 所示。

用户可以选择下面 5 种安装类型。

(1) Developer Default：安装开发 MySQL 应用程序所需的所有产品，例如 MySQL Server、MySQL Workbench、MySQL 连接器等。

(2) Server only：只安装 MySQL Server 产品。

(3) Client only：只安装 MySQL 客户端产品，例如 MySQL Workbench、MySQL 连接

器、示例/教程和文档。

(4) Full：完全安装。

(5) Custom：自定义安装。

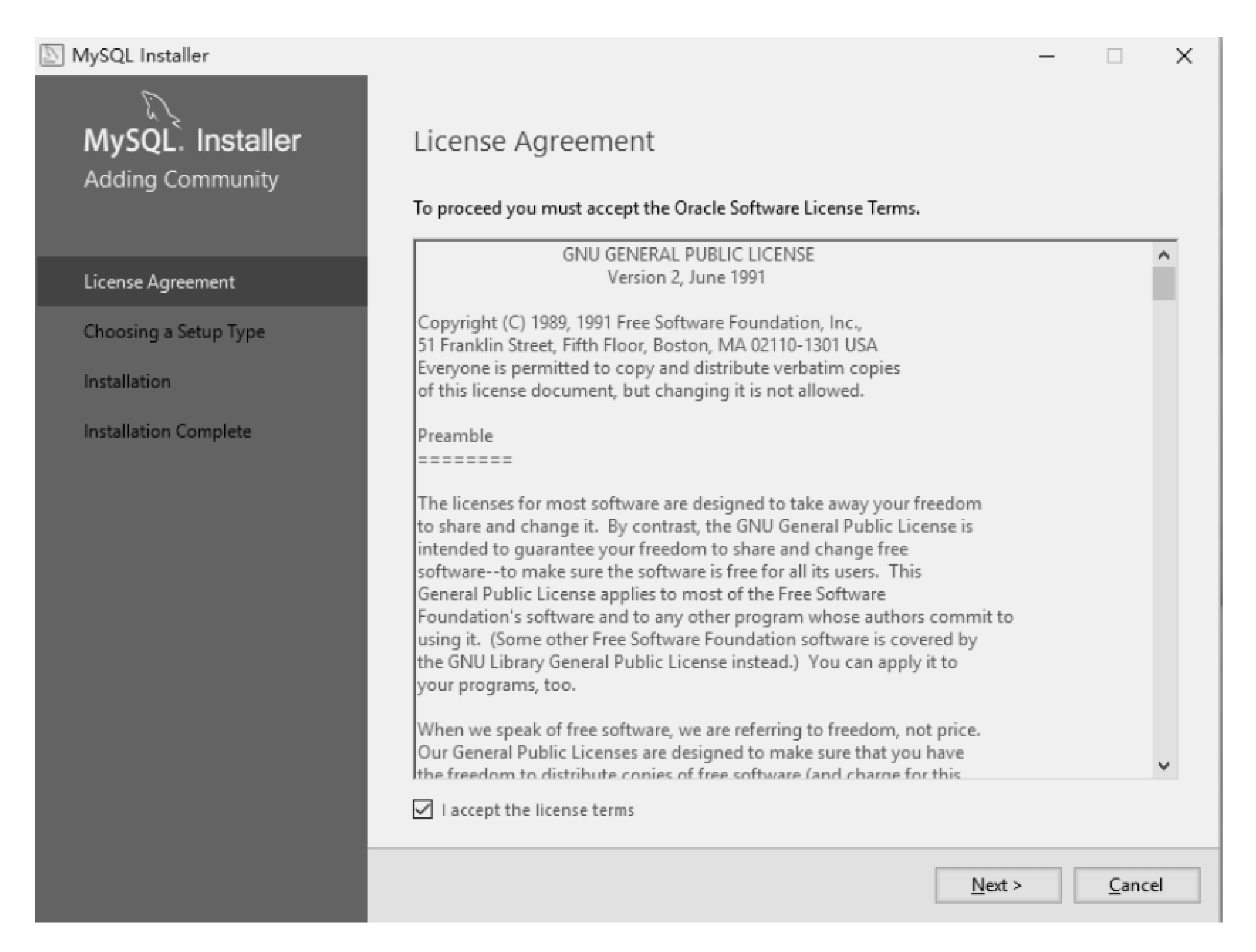


图 7-9　MySQL Installer 安装向导

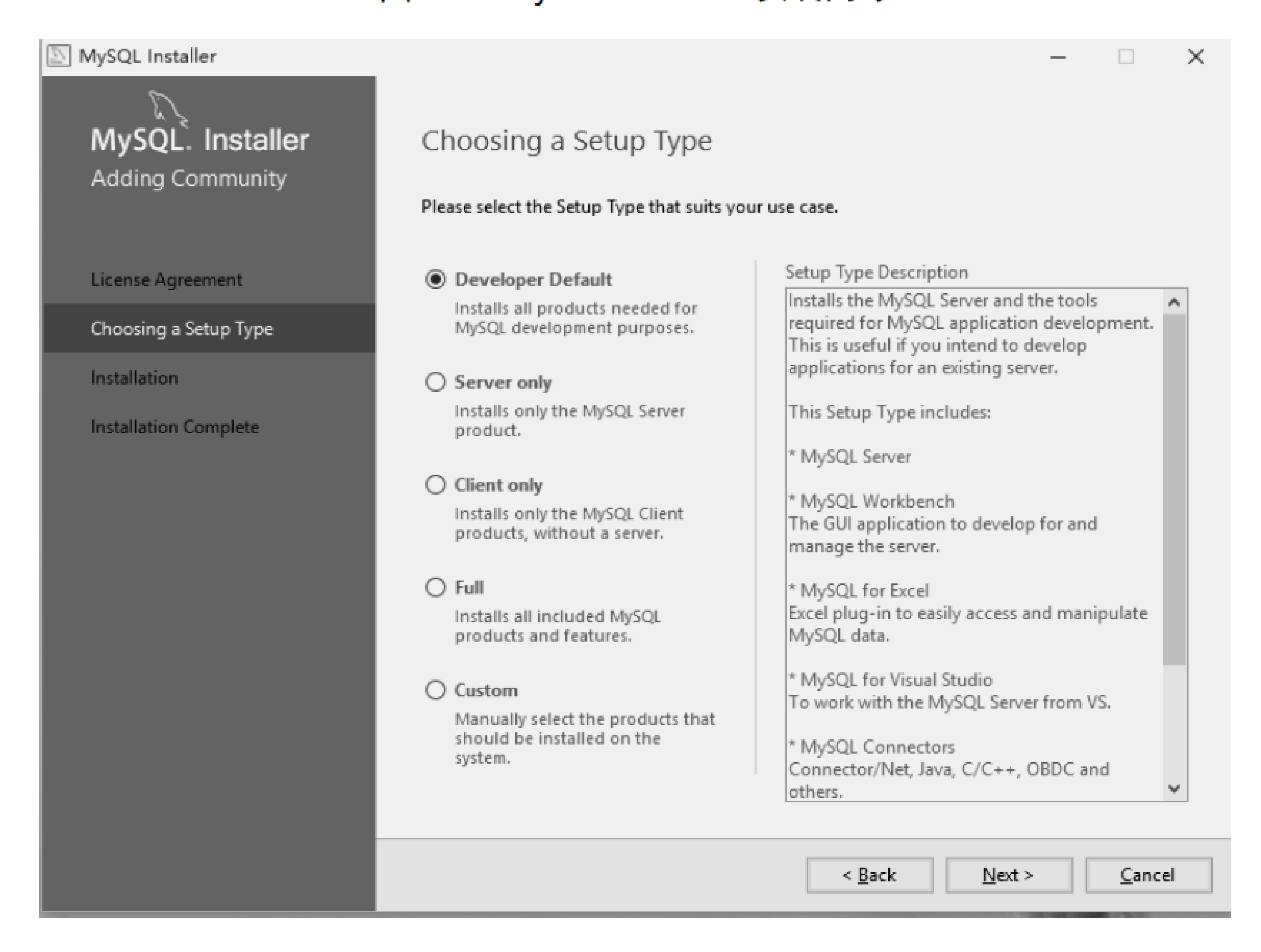


图 7-10　配置安装类型界面

选择 Full 进行完全安装，默认安装路径为 C:\Program Files\MySQL\MySQL。单击 Next 按钮，打开检查组件窗口，如图 7-11 所示。如果安装过程中提示缺少组件，则安装

组件后再尝试安装 MySQL 数据库。在本节中选择安装了 MySQL 可视化工具 MySQL Workbench 6.3.8，因为下面一些例题的结果要使用该工具进行展示。如果有些产品不需要用的话，则不需要安装这些额外组件，直接单击 Next 按钮就可以了。这时，会弹出一个窗口，忽略它，直接单击 Yes 按钮，然后进入安装窗口，如图 7-12 所示，单击 Execute 按钮开始安装。

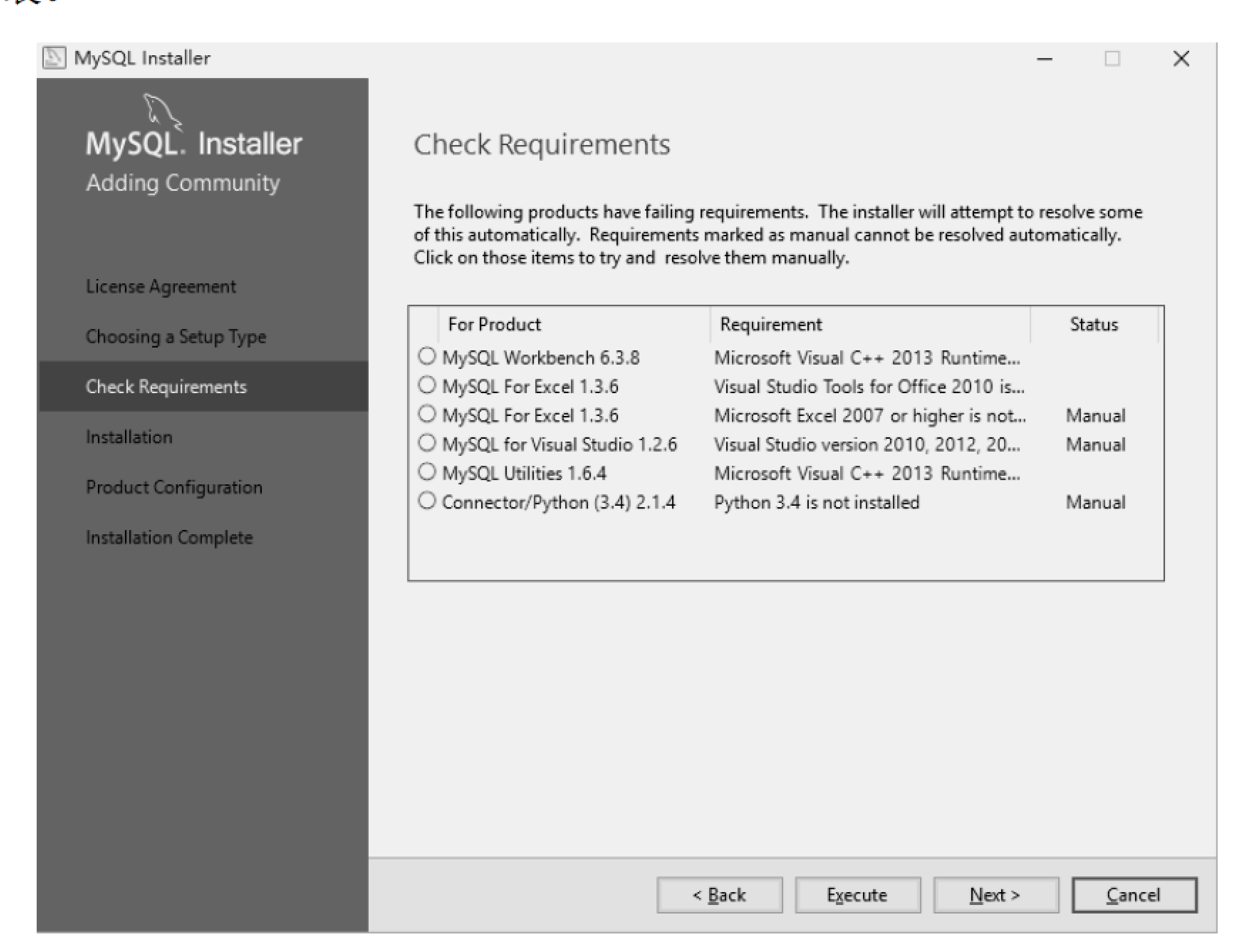


图 7-11　检查组件窗口

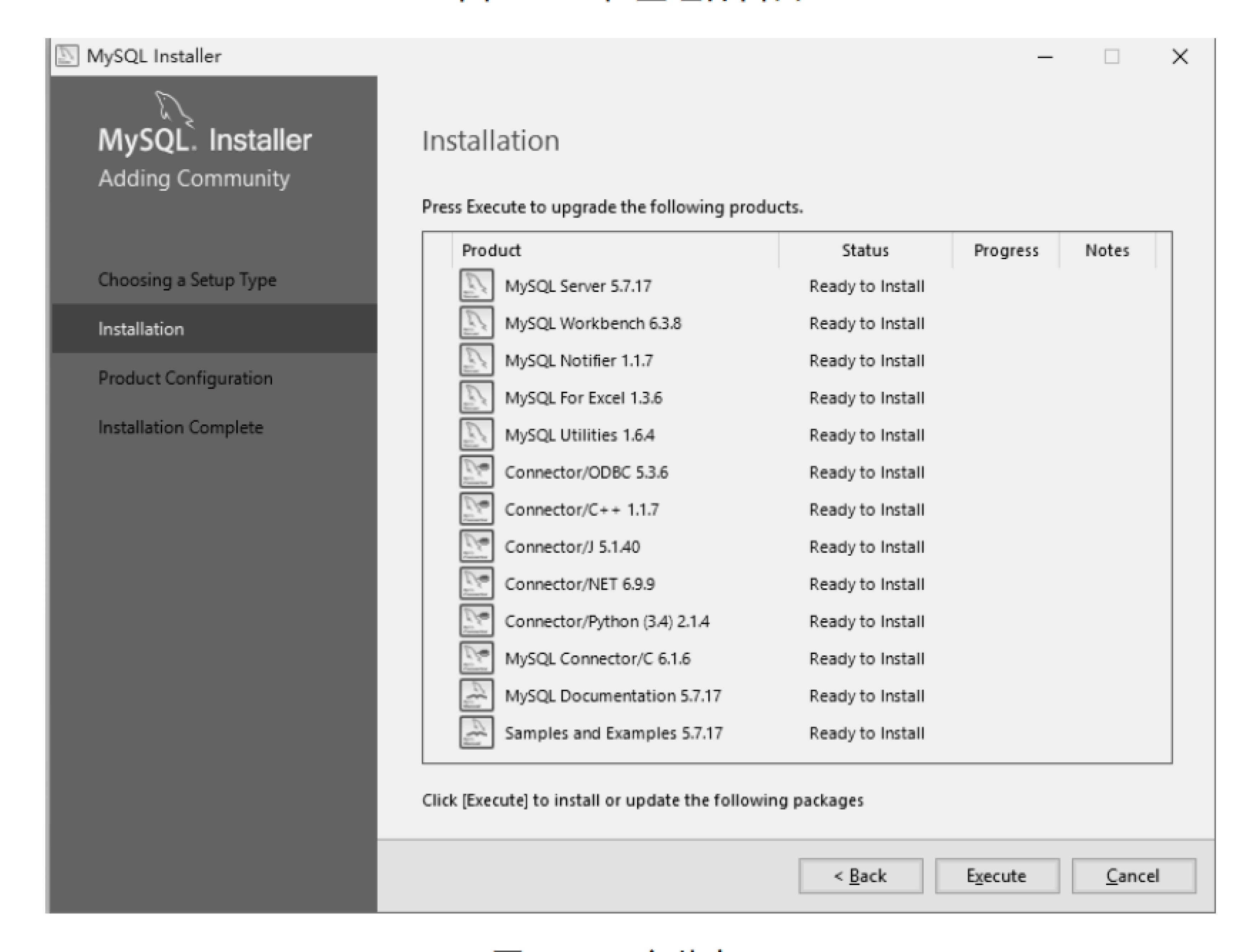


图 7-12　安装窗口

等待安装完成后，单击 Next 按钮，进入如图 7-13 所示的窗口，对 MySQL Server 进行配置。

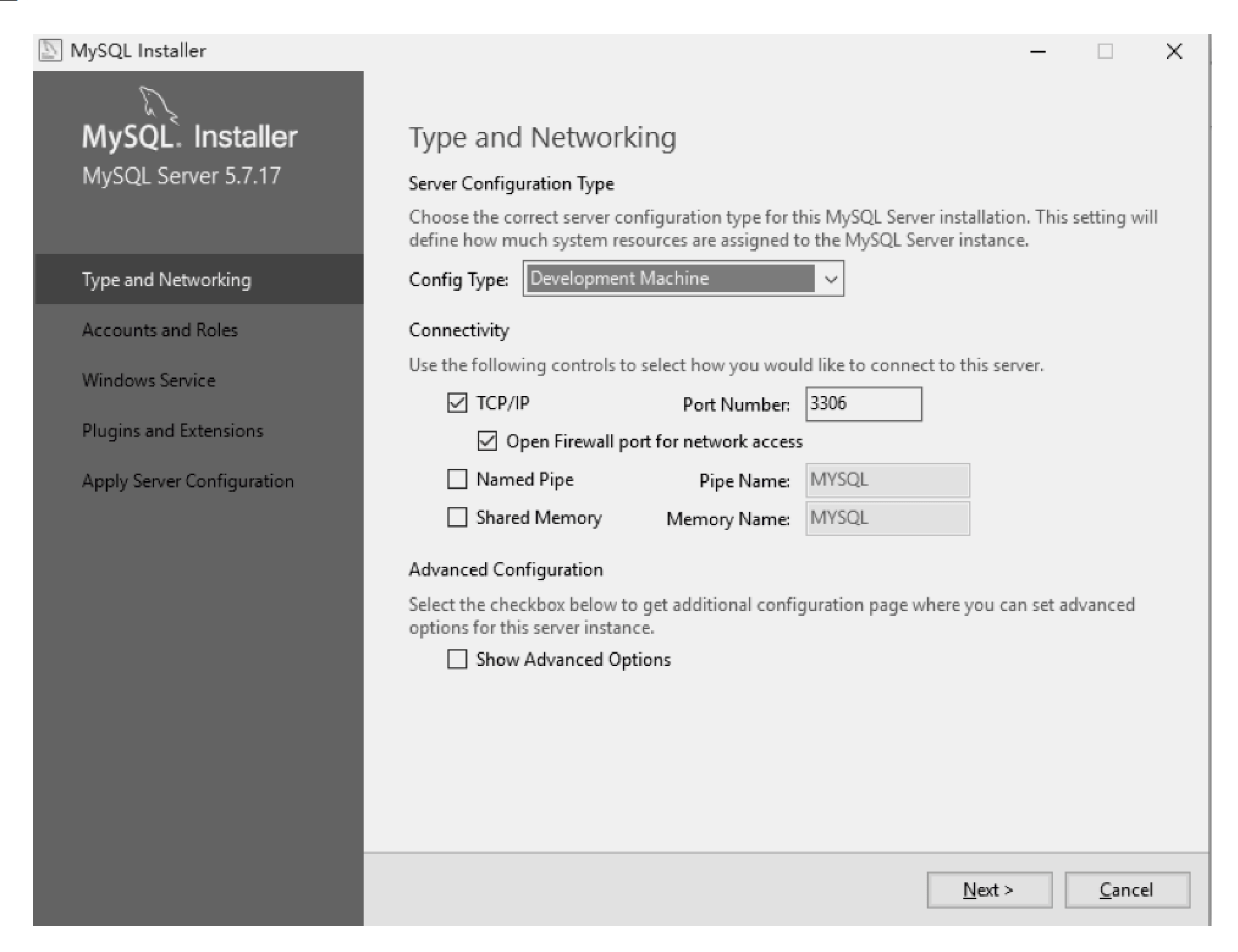


图 7-13　配置 MySQL Server 的窗口

用户可视需要，选择下面三种服务器类型之一。

Developer Machine：开发测试类型，主要针对个人使用，占用系统资源较少。

Server Machine：服务器类型，占用系统资源较多。若将计算机作为其他应用程序的服务器，如 FTP、E-mail、Web 服务器等，则可以将数据库配置为此类型。

Dedicated Machine：专门的 MySQL 数据库服务器，只用作 MySQL 服务器，不运行其他程序耗用系统所有可用资源。

根据需要，在本节中，我们选择 Server Machine 进行安装，MySQL 的 TCP 默认端口为 3306，如果仅仅是本地软件使用，不需要用网络来连接 MySQL 的话，也是可以不选择的。Named Pipe 是局域网用的协议，如果需要可以勾选。Shared Memory 协议是仅可以连接到同一台计算机上运行的 SQL Server 实例。接下来单击 Next 按钮，设置 MySQL 数据库管理员用户 root 的密码。设置好后，单击打开 Windows Server 窗口，如图 7-14 所示，设置 Windows 系统服务和插件扩展的选项。

至此，MySQL 数据库的安装接近完成，单击 Next 按钮，接下来就是一些检查或开启状态窗口，按默认单击 Next 按钮就可以了。安装完成后，配置 MySQL 环境变量，将 MySQL 的安装路径添加到环境变量的 PATH 变量中即可。

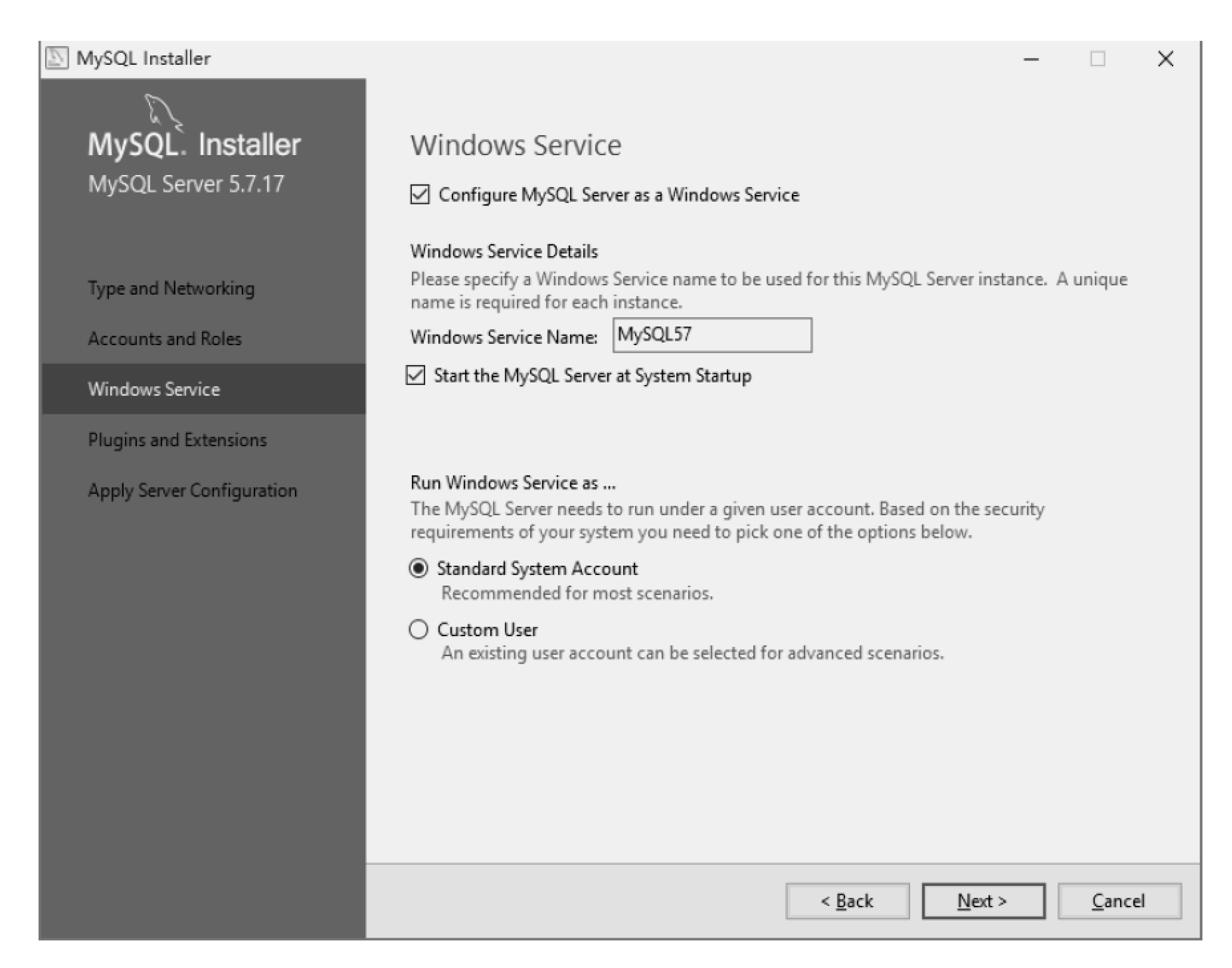


图 7-14　Windows Server 窗口

7.3.2　MySQL 数据类型

在 MySQL 中合理定义数据字段的类型对数据库的优化是非常重要的。MySQL 支持多种数据类型，主要分为三大类：数值类型、字符串(字符)类型、日期和时间类型。

1. 数值类型

MySQL 支持的数值数据类型如表 7-9 所示。

表 7-9　数值数据类型

数据类型	说　明
TINYINT	很小的整数值，有符号的范围为[-128，127]，无符号的范围为[0，255]
SMALLINT	小整数值，有符号的范围为[-32768，32767]，无符号的范围为[0，65535]
MEDIUMINT	中等大小的整数值，有符号的范围为[-8388608，8388607]，无符号的范围为[0，16777215]
INT 或 INTEGER	大整数值，有符号的范围为[-2147483648，2147483647]，无符号的范围为[0，4294967295]
BIGINT	极大整数值，有符号的范围为[-9233372036854775808，9223372036854775807]，无符号的范围为[0，18446744073709551615]
FLOAT(M,D)	单精度浮点值，其中 M 表示该值一共显示 M 位整数，D 表示其中 D 位位于小数点后面

续表

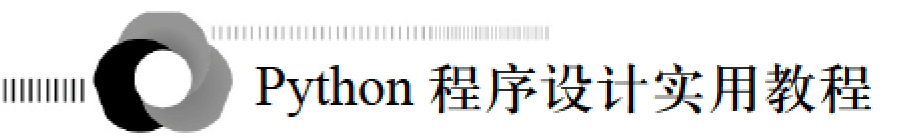
数据类型	说　明
TINYINT	很小的整数值，有符号的范围为[−128，127]，无符号的范围为[0，255]
SMALLINT	小整数值，有符号的范围为[−32768，32767]，无符号的范围为[0，65535]
MEDIUMINT	中等大小的整数值，有符号的范围为[−8388608，8388607]，无符号的范围为[0，16777215]
INT 或 INTEGER	大整数值，有符号的范围为[−2147483648，2147483647]，无符号的范围为[0，4294967295]
BIGINT	极大整数值，有符号的范围为[−9233372036854775808，9223372036854775807]，无符号的范围为[0，18446744073709551615]
FLOAT(M,D)	单精度浮点值，其中 M 表示该值一共显示 M 位整数，D 表示其中 D 位位于小数点后面
DOUBLE(M,D)	双精度浮点值，其中 M 表示该值一共显示 M 位整数，D 表示其中 D 位位于小数点后面
DECIMAL(M,D)	定点数，其中 M 表示十进制数字总的个数，D 表示小数点后面数字的位数，M 的默认取值为 10，D 默认取值为 0
BIT(M)	位字段值，允许存储 M 位值，M 范围为[1,64]，默认为 1

2. 字符串类型

MySQL 支持的字符串数据类型如表 7-10 所示。

表 7-10　字符串数据类型

数据类型	说　明
CHAR(N)	固定长度的字符串，N 为存储长度
VARCHAR(N)	可变长度的字符串，N 为最大存储长度
BINARY(N)	类似于 CHAR 类型，但不同的是 BINARY 存储的是二进制字节字符串，N 为存储长度
VARCHAR(N)	类似于 VARCHAR 类型，但不同的是 VARCHAR 存储的是二进制字节字符串，N 为存储长度
BLOB	二进制大对象，可以容纳可变数量的数据。包括 TINYBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB 种类型，这 4 种类型可容纳值的最大长度不同
TEXT	大文本类型，有 4 种 TEXT 类型，即 TINYTEXT、TEXT、MEDIUMTEXT 和 LONGTEXT，对应 4 种 BLOB 类型
ENUM	枚举类型
SET	集合类型



3. 日期和时间类型

MySQL 支持的日期和时间数据类型如表 7-11 所示。

表 7-11　日期和时间数据类型

数据类型	说　明
DATE	日期值，例如 2017-04-06
TIME	时间值，例如 14:20:30
YEAR	年份值，默认为四位年份值
DATETIME	日期和时间，例如 2017-04-06 14:20:30
TIMESTAMP	时间戳，自动存储记录修改的时间， 用于 INSERT 或 UPDATE 操作时记录日期和时间

7.3.3　MySQL 的基本操作

1. 登录到 MySQL

首先启动 MySQL 服务，当 MySQL 服务已经运行时，打开 Windows 命令提示符窗口输入以下格式的命令：

```
mysql -h 主机名 – u 用户名 -p （若登录当前计算机，则“-h 主机名”可以省略）
```

例如，在命令行下输入如下命令并回车：

```
mysql -u root -p
```

会有 Enter password 提示，此时应输入密码，若密码正确，则可以看到如图 7-15 所示的提示。



图 7-15　登录 MySQL

2. 创建数据库

使用 CREATE DATABASE 语句来完成数据库的创建，语句格式如下：

```
CREATE DATABASE IF NOT EXISTS 数据库名;
```

若使用关键字 IF NOT EXISTS，当指定的数据库存在时，不创建数据库。若不使用关键字 IF NOT EXISTS，并且指定的数据库存在，将产生错误。

【例 7-11】创建数据库 Test_db：

```
CREATE DATABASE IF NOT EXISTS Test_db;
```

3. 删除数据库

使用 DROP DATABASE 语句可以删除数据库，基本语法格式如下：

```
DROP DATABASE 数据库名;
```

使用 SHOW DATABASES 语句可显示所有数据库。

4. 创建数据库表

使用 CREATE TABLE 语句创建数据库表，基本语法格式如下：

```
CREATE TABLE 表名
(
    列名 1   数据类型  字段属性,
    列名 2   数据类型  字段属性,
    ...
    列名 n   数据类型  字段属性
);
```

常用的字段属性如表 7-12 所示。

表 7-12　常用的字段属性

字段属性	描　述
PRIMARY KEY	设置指定列为主键，用于确保记录的唯一性
AUTO_INCREMENT	设置指定列为自动增加列，每个新插入的记录赋值为上一次插入的 ID+1
INDEX	为指定列创建索引，加速数据库查询
NOT NULL	设置指定列的值不允许为空
NULL	设置指定列的值允许为空
DEFAULT	设置指定列的默认值
UNIQUE	设置指定列所有值除 NULL 外都不相同
BINARY	设置指定列以区分大小写方式排序

【例 7-12】使用 CREATE TABLE 语句创建教师信息表 Teacher，表 Teacher 的结构如表 7-13 所示。

表 7-13　表 Teacher 的结构

字 段 名	描　述	数据类型	字段属性
TNo	教师编号	INT	主键，自动递增列
TName	姓名	VARCHAR(50)	不允许为空
TSex	性别	CHAR(2)	默认值为“男”
TAge	年龄	INT	不允许为空
TTitle	职称	VARCHAR(50)	不允许为空
TDe	任职学院	VARCHAR(50)	不允许为空

```
CREATE TABLE Teacher (
    TNo INT AUTO INCREMENT PRIMARY KEY,
    TName VARCHAR(50) NOT NULL,
    TSex CHAR(2) DEFAULT'男',
    TAge INT NOT NULL,
    TTitle VARCHAR(50) NOT NULL,
    TDe VARCHAR(50) NOT NULL
);
```

执行此命令前可以使用 USE 语句选择一个所要操作的数据库，USE 语句可以不加分号，它的基本语法格式为：

```
USE 要操作的数据库名
```

5. 修改表结构

使用 ALTER TABLE 语句修改表结构。

(1) 向表中添加列，其基本语法格式为：

```
ALTER TABLE 表名 ADD 列名 数据类型 列属性;
```

【例 7-13】使用 ALTER TABLE 语句在表 Teacher 中增加所属学院编号列，列名为 DNo，数据类型为 INT，列属性为“不允许为空”：

```
ALTER TABLE Teacher ADD DNo INT NOT NULL;
```

(2) 修改列属性，其基本语法格式为：

```
ALTER TABLE 表名 MODIFY 列名 新数据类型 新列属性;
```

【例 7-14】使用 ALTER TABLE 语句在表 Teacher 中修改 DNo 列，将数据类型修改为 CHAR(5)，列属性为“允许为空”：

```
ALTER TABLE Teacher MODIFY DNo CHAR(5) NULL;
```

(3) 删除列，其基本语法格式为：

```
ALTER TABLE 表名 DROP COLUMN 列名;
```

【例 7-15】删除表 Teacher 中的 DNo 列，具体命令如下：

```
ALTER TABLE Teacher DROP COLUMN DNo;
```

6. 删除表

使用 DROP TABLE 语句删除数据库中的表，其基本语法格式为：

```
DROP TABLE 表名;
```

7. 插入数据

使用 INSERT 语句向表内插入数据，语法格式为(列与值必须一一对应)：

```
INSERT INTO 表名 (列名 1, 列名 2,...,列名 n)
VALUES(值 1,值 2,...,值 n);
```

【例 7-16】使用 INSERT 语句参照表 7-14 向表 Teacher 中插入数据(由于设置了字段 TNo 为 AUTO_INCREMENT 属性，在这并不需要指定字段 TNo 的值)。

表 7-14　表 Teacher 中插入的数据

TName	TSex	TAge	TTitle	TDe
李丽	女	55	教授	计算机科学与软件学院
王阳	男	40	副教授	纺织学院
赵亮	男	35	讲师	纺织学院
张亮	男	37	副教授	管理学院
王丽	女	43	讲师	计算机科学与软件学院
张阳	女	48	教授	经济学院

命令如下：

```
INSERT INTO Teacher (TName,TSex,TAge,TTitle,TDe) VALUES ('李丽','女',55,
'教授','计算机科学与软件学院');
INSERT INTO Teacher (TName,TAge,TTitle,TDe) VALUES ('王阳',40,'副教授',
'纺织学院');
INSERT INTO Teacher (TName,TAge,TTitle,TDe) VALUES ('赵亮',35,'讲师',
'纺织学院');
INSERT INTO Teacher (TName,TAge,TTitle,TDe) VALUES ('张亮',37,'副教授',
'管理学院');
INSERT INTO Teacher (TName,TSex,TAge,TTitle,TDe) VALUES ('王丽','女',43,
'讲师','计算机科学与软件学院');
INSERT INTO Teacher (TName,TSex,TAge,TTitle,TDe) VALUES ('张阳','女',48,
'教授','经济学院');
```

在可视化工具 MySQL Workbench 中可以直观地看到例 7-16 使用 INSERT 语句所创建的表 Teacher，如图 7-16 所示。

TNo	TName	TSex	TAge	TTitle	TDe
1	李丽	女	55	教授	计算机科学与软件学院
2	王阳	男	40	副教授	纺织学院
3	赵亮	男	37	讲师	纺织学院
4	张亮	男	37	副教授	管理学院
5	王丽	女	43	讲师	计算机科学与软件学院
6	张阳	女	48	教授	经济学院
NULL	NULL	NULL	NULL	NULL	NULL

图 7-16　例 7-16 所创建的表 Teacher

8. 修改数据

使用 UPDATE 语句修改表中的数据，语句的一般格式为：

```
UPDATE 表名 SET 列名 1=值 1, 列名 2=值 2, ..., 列名 n=值 n
WHERE 修改条件表达式;
```

该语句的功能是修改表中满足 WHERE 子句条件的记录。其中 SET 子句用于指定修改方法，即列 1 的值被设置为值 1，列 2 的值被设置为值 2，列 n 的值被设置为值 n。如果省略 WHERE 子句，则表中所有的记录都将被修改。

【例 7-17】使用 UPDATE 语句修改表 Teacher，将赵亮的年龄改为 34：

```
UPDATE Teacher SET TAge=34 WHERE TName = '赵亮';
```

9. 删除数据

使用 DELETE 语句删除表中的数据，语法格式如下：

```
DELETE FROM 表名 WHERE 删除条件表达式;
```

DELETE 语句的功能是将指定表中满足 WHERE 子句条件的所有记录删除。如果没有提供 WHERE 子句，则 DELETE 语句将删除表中的所有记录，但表的结构仍在，也就是说，DELETE 语句删除的是表中的数据。

10. 使用 SELECT 语句查询数据

(1) 查询指定列。

语句的一般格式为：

```
SELECT 列名 1, 列名 2, ..., 列名 n FROM 表名;
```

若查询全部列，可用“*”代替“列名 1, 列名 2, …, 列名 n”。如下面的例子。

【例 7-18】查询表 Teacher 中的所有信息：

```
SELECT * FROM Teacher;
```

查询结果如图 7-17 所示。

(2) 给列指定别名。

有两种格式：列名 别名；列名 AS 别名。如下面的例子。

【例 7-19】查询表 Teacher 中的 TName、TTitle，要求显示中文列名：

```
SELECT TName '姓名', TTitle '职称' FROM Teacher;
```

或者：

```
SELECT TName AS '姓名', TTitle AS '职称' FROM Teacher;
```

查询结果如图 7-18 所示。

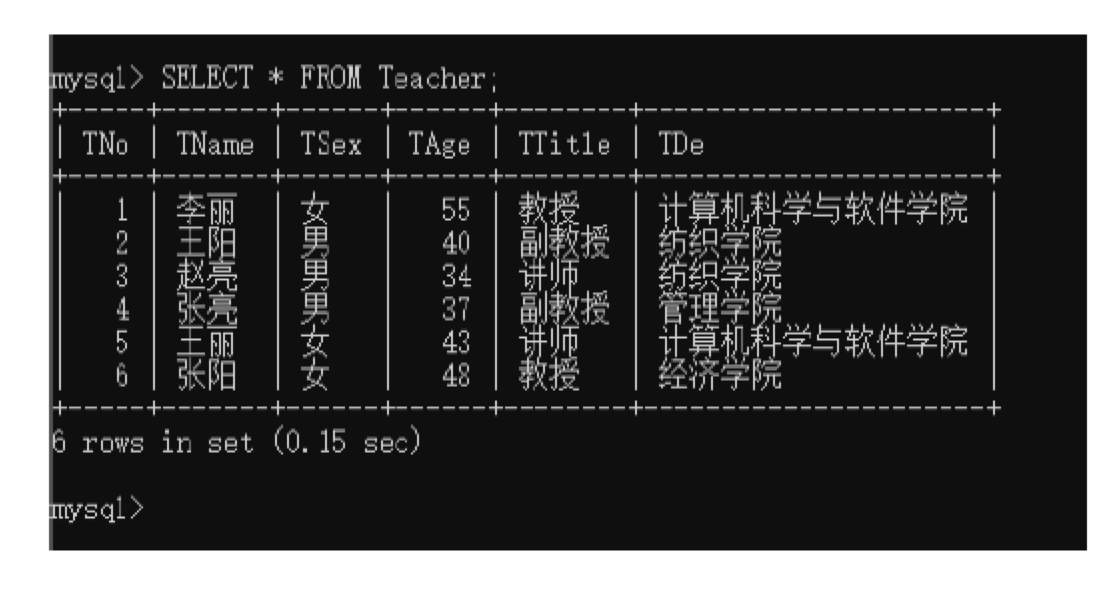


图 7-17　例 7-18 的查询结果

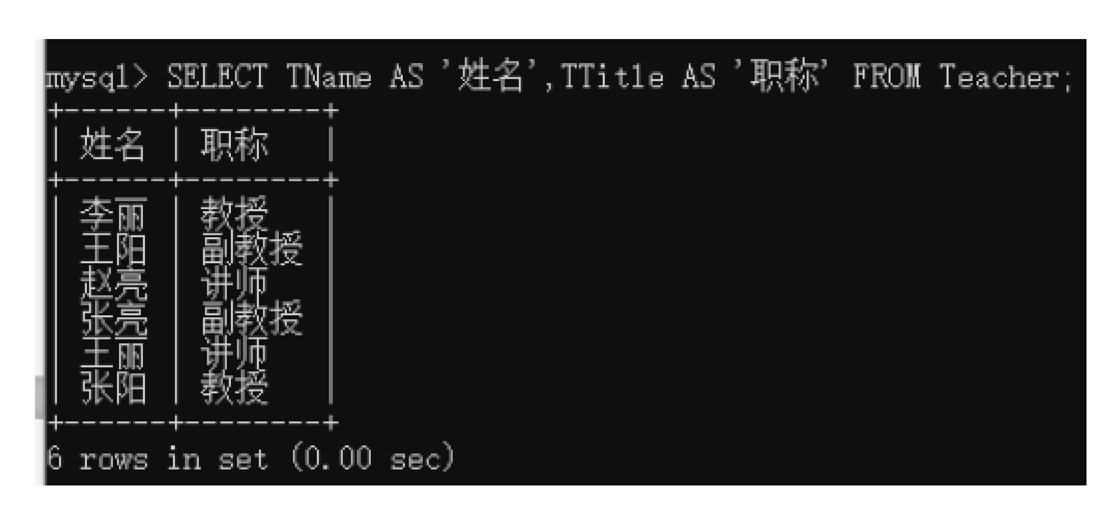


图 7-18　例 7-19 的查询结果

(3)　消除取值重复行。

使用关键字 DISTINCT 可消除取值重复的行。如下面的例子。

【例 7-20】 查询所有教师的职称情况：

```
SELECT DISTINCT TTitle FROM Teacher;
```

查询结果如图 7-19 所示。

```
mysql> SELECT DISTINCT TTitle FROM Teacher;
+--------+
| TTitle |
+--------+
| 教授   |
| 副教授 |
| 讲师   |
+--------+
3 rows in set (0.16 sec)

mysql>
```

图 7-19　例 7-20 的查询结果

(4)　设置查询条件。

WHERE 子句可以指定返回结果的查询条件。

WHERE 子句中常用的查询条件如表 7-15 所示。

表 7-15　WHERE 子句常用的查询条件

查询条件	谓　词	描　述
比较	=	等于，例如 TAge=40
	>	大于，例如 TAge>2
	<	小于，例如 TAge<2
	>=	大于等于，例如 TAge>=2
	<=	小于等于，例如 TAge<=2
	!=或<>	不等于，例如 TAge!=2 或 TAge<>2
	!>	不大于，例如 TAge!>2
	!<	不小于，例如 TAge!<2

续表

查询条件	谓　词	描　述
确定范围	BETWEEN AND	判断指定列的属性值是否在指定范围内，例如 TAge BETWEEN 30 AND 50
	NOT BETWEEN AND	判断指定列的属性值是否不在指定范围内，例如 TAge NOT BETWEEN 30 AND 50
确定集合	IN	判断指定列的属性值是否属于指定集合，例如 TAge IN(30,37,45,55)
	NOT IN	判断指定列的属性值是否不属于指定集合，例如 TAge NOT IN(30,37,45,55)
字符匹配	LIKE	判断指定列的属性值是否与匹配字符串相匹配，匹配字符串可以是一个完整的字符串，也可含有通配符%和_(%代表任意长度的字符串，_代表任意单个字符)，例如 TName LIKE ‘张%’
	NOT LIKE	判断指定列的属性值是否与匹配字符串不相匹配，例如 TName NOT LIKE ‘张%’
空值	IS NULL	判断指定列的属性值是否为空，例如 TAge IS NULL
	IS NOT NULL	判断指定列的属性值是否不为空，例如 TAge IS NOT NULL
多重条件(逻辑运算)	AND(&&)	逻辑与，查询满足所有条件的记录
	OR(||)	逻辑或，查询满足任一条件的记录
	NOT(!)	逻辑非，查询不满足表达式的记录

(5) 对查询结果进行排序。

通过在 SELECT 语句中使用 ORDER BY 子句，可以根据指定列对查询结果进行排序。ORDER BY 子句默认的排序顺序为升序(ASC)，若要按降序排序，必须指明 DESC 选项。如下面的例子。

【例 7-21】查询表 Teacher 中全体男教师的信息，要求查询结果按照年龄降序排列：

```
SELECT * FROM Teacher WHERE TSex='男' ORDER BY TAge DESC;
```

查询结果如图 7-20 所示。

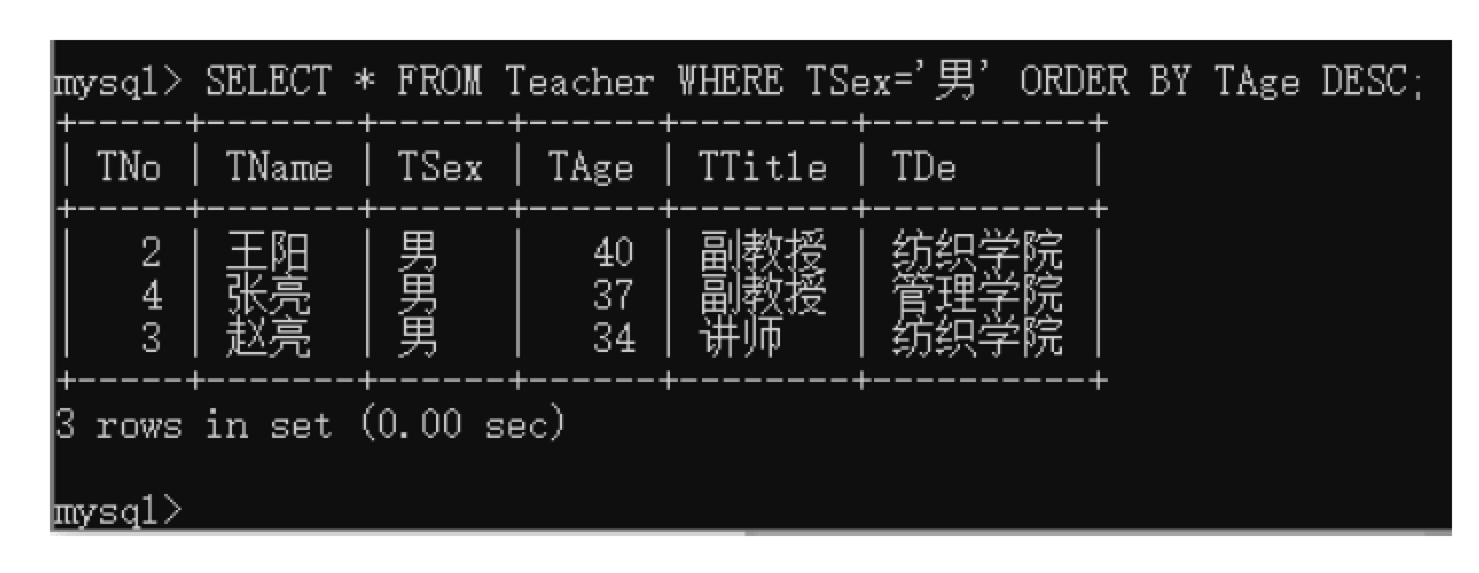


图 7-20　例 7-21 的查询结果

(6) 使用统计函数。

在 SELECT 语句中使用统计函数，可以对指定列进行统计。MySQL 中常用的统计函数主要有以下 5 种。

① MAX()：统计指定列的最大值。

② MIN()：统计指定列的最小值。

③ SUN()：统计指定列的总和。

④ AVG()：统计指定列的平均值。

⑤ COUNT()：统计记录个数。

当聚集函数遇到空值时，除了 COUNT(*)外，其他的函数都会忽略空值，只处理非空值。如果在统计函数中使用关键字 DISTINCT，则表示在统计时先消除指定列取重复值的记录，然后再进行统计；如果不指定关键字 DISTINCT 或指定关键字 ALL(ALL 为默认值)，则表示不取消指定列重复值的记录。

【例 7-22】统计表 Teacher 中所有教师的平均年龄：

```
SELECT AVG(TAge) FROM Teacher;
```

为了便于理解，可以对统计列取列名，语句修改如下：

```
SELECT AVG(TAge) '平均年龄' FROM Teacher;
```

统计结果如图 7-21 所示。

```
mysql> SELECT AVG(TAge) FROM Teacher;
+-----------+
| AVG(TAge) |
+-----------+
|   42.8333 |
+-----------+
1 row in set (0.12 sec)

mysql> SELECT AVG(TAge) '平均年龄' FROM Teacher;
+----------+
| 平均年龄 |
+----------+
|  42.8333 |
+----------+
1 row in set (0.02 sec)
```

图 7-21　例 7-22 的查询结果

(7) 分组统计。

在 SELECT 语句中使用 GROUP BY 子句，可用来对查询结果进行分组，并对每组数据进行汇总统计。

在 SELECT 语句中使用 GROUP BY 子句，SELECT 子句中只能出现分组列的列名和统计函数。如下面的例子。

【例 7-23】统计表 Teacher 中各职称教师的人数：

```
SELECT TTitle, COUNT(*) '人数' FROM Teacher GROUP BY TTitle;
```

统计结果如图 7-22 所示。

```
mysql> SELECT TTitle,COUNT(*) '人数' FROM Teacher GROUP BY TTitle;
+--------+------+
| TTitle | 人数 |
+--------+------+
| 副教授 |    2 |
| 教授   |    2 |
| 讲师   |    2 |
+--------+------+
3 rows in set (0.02 sec)
```

图 7-22　例 7-23 的查询结果

若分组后要按一定条件对这些组进行筛选，最终只输出满足指定条件的组，则使用 HAVING 子句指定筛选条件。HAVING 子句与 WHERE 子句作用类似，但 HAVING 子句只能用于 GROUP BY 子句，WHERE 是用于在初始表中筛选查询；HAVING 子句中可以使用聚集函数，而 WHERE 则不能。

【例 7-24】将表 Teacher 中的所有男教师按职称分组，统计每组教师的平均年龄：

```
SELECT TTitle, AVG(TAge) '平均年龄' FROM Teacher WHERE TSex='男' GROUP BY
TTitle;
```

统计结果如图 7-23 所示。

```
mysql> SELECT TTitle,AVG(TAge) '平均年龄' FROM Teacher
 WHERE TSex='男' GROUP BY TTitle;
+--------+----------+
| TTitle | 平均年龄 |
+--------+----------+
| 副教授 |  38.5000 |
| 讲师   |  34.0000 |
+--------+----------+
2 rows in set (0.00 sec)
```

图 7-23　例 7-24 的查询结果

【例 7-25】将表 Teacher 中的教师按职称分组，统计平均年龄大于 40 岁的教师的职称类型：

```
SELECT TTitle, AVG(TAge) '平均年龄' FROM Teacher GROUP BY TTitle HAVING
AVG(TAge)>40;
```

统计结果如图 7-24 所示。

```
mysql> SELECT TTitle,AVG(TAge) '平均年龄' FROM Teacher
GROUP BY TTitle HAVING AVG(TAge)>40;
+--------+----------+
| TTitle | 平均年龄 |
+--------+----------+
| 教授   |  51.5000 |
+--------+----------+
1 row in set (0.07 sec)
```

图 7-24　例 7-25 的查询结果

7.3.4　使用 Python 操作 MySQL 数据库

(1) 使用 Python 操作 MySQL 数据库，需要安装 pymysql 模块，它是 Python 操作 MySQL 必不可少的模块。安装步骤如下。

① 在 Python 官网上下载管理包工具 ez_setup.py。下载地址为 http://pypi.python.org/pypi/ez_setup/。

② 打开 DOS 命令提示符窗口，切换到 Python 目录下运行下面的命令，安装 easy_install.exe 工具包：

```
Python ez_setup.py
```

③ 安装完成后，可在 Python 目录下的 Scripts 目录中看到 easy_install.exe。打开 DOS 命令提示符窗口，切换到 Python 的 Scripts 目录下运行下面的命令，安装 pymysql：

```
easy_install pymysql
```

(2) 安装好 pymysql 后，就可以用 Python 来操作 MySQL 数据库了，下面介绍具体的操作步骤。

① 导入 pymysql 模块。

执行下面的语句将导入 pymysql 模块：

```
import pymysql
```

② 连接数据库。

使用 connect()方法建立数据库的连接，里面可以指定参数：用户名，密码，主机等信息，语法格式如下：

```
数据库连接对象 = pymsql.connect(数据库服务器,用户名,密码,数据库名)
```

③ 创建游标。

要操作数据库，需要通过 cursor()方法来创建游标。语法格式如下：

```
游标对象=数据库连接对象.cursor()
```

④ 执行 SQL 语句。

通过游标对象操作 execute()方法可以执行 SQL 语句，返回值为受影响的行数。语法格式如下：

```
游标对象.execute(SQL 语句)
```

⑤ 使用游标查询数据。

使用游标对象的 execute()方法执行 SELECT 语句，可将查询结果保存在游标中，语法格式如下：

```
游标对象.execute(SELECT 语句)
```

使用游标对象的 fetchall()方法获取游标中所有的数据，并放到一个元组中，语法格式如下：

```
结果集元组 = 游标对象.fetchall()
```

【例 7-26】在数据库 test_db 中使用游标查询表 Teacher 中的数据：

```
import pymysql
co=pymysql.connect("localhost","root","1234","test db",charset="utf8")
cx=co.cursor()
```

```
cx.execute("SELECT * FROM Teacher")
a=cx.fetchall()
print(a)
cx.close()
co.close()
```

运行结果如下：

```
((1, '李丽', '女', 55, '教授', '计算机科学与软件学院'), (2, '王阳', '男', 40,
'副教授', '纺织学院'), (3, '赵亮', '男', 34, '讲师', '纺织学院'), (4, '张亮',
'男', 37, '副教授', '管理学院'), (5, '王丽', '女', 43, '讲师', '计算机科学与
软件学院'), (6, '张阳', '女', 48, '教授', '经济学院'))
>>>
```

⑥　数据库的提交：

```
数据库连接对象.commit()
```

用于提交对数据库的修改，将数据保存到数据库中。

⑦　关闭数据库连接。

关闭游标对象：

```
游标对象.close()
```

关闭数据库连接，释放资源：

```
数据库连接对象.close()
```

7.4　案例实训：管理信息系统的数据操作

本案例程序集数据的增、删、改、查于一体，类似于规模很小的管理信息系统，用一个菜单程序作为调用接口，以选择执行某一个功能，选择后调用相应的程序完成相应的功能。本案例程序如下：

```
import pymysql
co=pymysql.connect("localhost","root","1234","test db",charset="utf8")
cx=co.cursor()
def insert info():   #插入数据
    cx=co.cursor()
    pSNo=input("请输入学生学号：")
    cx.execute("select SNo from student where SNo='%s'"% pSNo)
    row=cx.fetchone()
    if row:
        print("该学号已存在，请重新输入：")
    else:
        pSName=input("请输入学生姓名：")
        pSClass=input("请输入学生班级：")
        cx.execute("insert into student(SNo,SName,SClass)
          values('%s','%s','%s')"% (pSNo,pSName,pSClass))
        co.commit()
        print("学生信息录入完毕。")
    co.commit()
    cx.close()
```

```
def search info():  #查询数据
    cx=co.cursor()
    pSNo= input("请输入学生学号:")
    cx.execute(
      "SELECT SNo,SName,SClass from student where SNo='%s'"% pSNo)
    row = cx.fetchone()
    if row:
        print("您所查询的学生信息为: ")
        print("学号:",row[0])
        print("姓名:",row[1])
        print("班级:",row[2]),"\n"
    else:
        print("没有查询该学号的学生信息! ")
    cx.close()
def update info():   #修改数据
    cx=co.cursor()
    pSNo=input("请输入学生学号:")
    cx.execute("SELECT SNo from student where  SNo='%s'"% pSNo)
    row = cx.fetchone()
    if row:
        colums=input("请输入修改的列名(SNo,SName,SClass):")
        value=input("请输入新值:")
        cx.execute(
        "update student set %s='%s' where SNo ='%s'"% (colums,value,pSNo))
        print("修改完毕! ")
    else:
        print("该学号不存在,请重新输入")
    co.commit()
    cx.close()
def delete info():  #删除数据
    cx=co.cursor()
    pSNo=input("请输入学生学号:")
    cx.execute("SELECT SNo from student where SNo = '%s'"% pSNo)
    row = cx.fetchone()
    if row:
        cx.execute("delete from student where SNo = '%s'"% pSNo)
        print("删除完毕! ")
    else:
        print("该学号不存在,请重新输入")
    co.commit()
    cx.close()
def menu():   #菜单目录
    print("1.信息录入")
    print("2.信息删除")
    print("3.信息修改")
    print("4.信息查询")
    print("5.退出! ")
def main():   #菜单目录选择
    while True:
        menu()
        x=input("输入您所选择的菜单号: ")
        print
        if x =='1':
```

```
            insert info()
            continue
        if x =='2':
            delete info()
            continue
        if x =='3':
            update info()
            continue
        if x =='4':
            search info()
            continue
        if x =='5':
            print("欢迎再次使用本系统！谢谢！")
            exit()
        else:
            print("输入的选项不存在，请重新输入！")
            continue
main()
```

本程序的添加、删除、修改、查询信息功能的运行结果如图 7-25 ~ 7-28 所示。

```
1.信息录入
2.信息删除
3.信息修改
4.信息查询
5.退出！
输入您所选择的菜单号：1
请输入学生学号：0004
请输入学生姓名：帆帆
请输入学生班级：一年级3班
学生信息录入完毕。
```

图 7-25　添加信息功能

```
1.信息录入
2.信息删除
3.信息修改
4.信息查询
5.退出！
输入您所选择的菜单号：2
请输入学生学号:0004
删除完毕！
1.信息录入
2.信息删除
3.信息修改
4.信息查询
5.退出！
```

图 7-26　删除信息功能

```
1.信息录入
2.信息删除
3.信息修改
4.信息查询
5.退出！
输入您所选择的菜单号：3
请输入学生学号:0003
请输入修改的列名（SNo,SName,SClass）:SClass
请输入新值:一年级3班
修改完毕！
1.信息录入
2.信息删除
3.信息修改
4.信息查询
5.退出！
```

图 7-27　修改信息功能

```
1.信息录入
2.信息删除
3.信息修改
4.信息查询
5.退出！
输入您所选择的菜单号：4
请输入学生学号:0004
您所查询的学生信息为：
学号：0004
姓名：帆帆
班级：一年级3班
```

图 7-28　查询信息功能

本 章 小 结

Python 支持多种数据库，其中包括 SQLite 数据库和 MySQL 数据库。本章简要介绍了数据库的一些基本概念和两种关系型数据库——SQLite 数据库与 MySQL 数据库的基本使用操作，以及如何通过 sqlite3 模块和 pymsql 模块分别操作这两种数据库。事实上，Python 标准数据库接口为 Python DB-API，它支持多种数据库，不同的数据库需要不同的 DB-API 模块，因篇幅所限，本章没有介绍。

习 题

1. 填空题

(1) 根据数据存储模型，可将数据库分为( )、( )、( )、面向对象数据库等，SQLite 数据库和 MySQL 数据库都属于( )数据库。

(2) SQLite 数据库中，字段属性 PRIMARY KEY 用来设置指定字段为( )，以确保记录的( )。

(3) 使用 SELECT 语句查询数据时，可以使用关键字( )来消除取值重复行。

(4) Python 连接 MySQL 数据库时需要单独下载( )模块，并执行语句( )来导入该模块。

(5) HAVING 子句与 WHERE 子句作用类似，都是指定筛选条件，但 HAVING 子句只能用于( )子句，WHERE 是用于在初始表中筛选查询；HAVING 子句中可以使用( )，而 WHERE 则不能。

2. 选择题

(1) 数据库是长期存储在计算机内、有组织、统一管理的相关( )。

A. 文件的集合 B. 数据的集合 C. 数值的集合 D. 程序的集合

(2) 下列哪些是关系型数据库？

① Redis ② DB2 ③ MySQL ④ MongoDB ⑤ Oracle ⑥ HBase

A. ②③⑤ B. ①②⑥ C. ②④⑤ D. ①⑤⑥

(3) 下列属于 MySQL 的日期和时间数据类型的是( )。

① INT ② DATE ③ INYINT ④ TIMESTAMP ⑤ YEAR ⑥ BIT

A. ①②⑥ B. ①②③ C. ②④⑤ D. ①⑤⑥

(4) 下列哪种方法可以提交事务？

A. execute() B. fetchall() C. connect() D. commit()

(5) 使用下面哪个语句可以更新数据？

A. CREAT B. UPDATE C. DELETE D. INSERT INTO

3. 问答题

(1) 什么是数据库、数据库管理系统和数据库系统？简述三者的不同之处。

(2) 简单介绍 SQLite 数据库以及 sqlite3 模块提供的数据库访问方法。

4. 实验操作题

参照本章 7.2.5 小节的示例，用 Python 操作 SQLite 数据库，实现以下任务。

(1) 创建数据库和表，数据库名为 mydatabase，表名为 user，表中包含三列：id、name 和 tel，其中 id 为主键，name 不允许为空。

(2) 编写对表 user 中的数据进行插入、修改和删除的程序。

(3) 查询表中数据。



第 8 章

Web 开发

本章要点

(1) Web 应用的工作方式。
(2) MVC 设计模式。
(3) CGI 通用网关接口。
(4) 使用模板快速生成 Web 页面。

学习目标

(1) 掌握 Web 应用的基本工作方式。
(2) 掌握 MVC 设计模式的具体内容。
(3) 了解 CGI 的相关知识。
(4) 掌握使用 Python 建立 Web 服务器的方法。
(5) 掌握使用模板快速生成 HTML 页面的方法。

Python 语言得益于其强大的标准库和第三方库的支持，使其可以很好地应用于 Web 软件开发。同时，由于它可以很好地支持最新的 XML 技术并和多种语言结合使用，也使其在 Web 开发方向的应用越来越广泛。本章将主要介绍如何使用 Python 进行 Web 开发。

8.1 将程序放在 Web 上运行

顺利地学习了前面的知识，相信读者已经能够使用 Python 语言编写一些简单的代码，用以解决工作或学习中遇到的问题了。试想，如果编写了一个得意的小工具，如何能让更多的人一起来分享这种方便呢？当对它进行功能升级后，如何让所有使用它的用户都可以马上体验到最新的版本？如何让输入输出有更好的用户体验？

使用 Python 语言的 Web 开发功能，将开发的程序放在 Web 服务器上运行，可以很好地解决上述问题。而且相对于代码裸露的单机版本，一个基于 Web 的应用会有如下的众多优点。

(1) 轻松访问，可以随时随地地通过浏览器访问你的应用。

(2) 跨平台访问，可以通过多种媒体工具访问你的应用，使你的应用不只是运行在单一的设备上。

(3) 增加用户，可以让更多的人方便地使用你的应用，同时更好地了解用户的需求。

(4) 同步更新，可以让所有人即刻用到最新的版本。

(5) 良好的界面，运用 Web 技术为程序编写一个舒服的界面，让代码不再枯燥。

8.1.1 Web 应用的工作方式

在学习编写一个 Web 应用之前，需要了解一下 Web 应用是如何在服务器上工作的。

不论我们在 Web 上进行什么操作，都会用到请求和响应。首先是用户通过 Web 浏览器发送一个包含交互操作(输入一个 URL、选择一个链接或者提交一个表单)的 Web 请求到 Web 服务器上。Web 服务器会根据请求的报文信息将需要的 HTML、CSS、JS 等文件生成

一个 Web 响应，并发回给 Web 浏览器。整个过程大致分为 4 个阶段。

(1) 用户从 Web 浏览器输入一个 URL、选择一个链接或者提交一个表单。

(2) Web 浏览器将用户的操作转换为一个 Web 请求，并通过网络通信将这个请求发送给相应的 Web 服务器。

(3) Web 服务器收到 Web 浏览器发送来的请求后，可能会进行两种操作：①如果 Web 请求的是静态内容(Static Content)，Web 服务器就会在本地找到这些资源(例如 HTML 文件、图片或者存储在 Web 服务器上的其他文件)，并将其内容封装到 HTTP 消息体中，以消息体的形式返回给 Web 服务器；②如果请求的是动态内容(Dynamic Content)，也就是说，内容是需要动态加载的(这样做的好处是可以与后台数据库进行交互操作)，那么服务器就会通过一个应用程序来生成 Web 响应，并发送给 Web 浏览器(服务器端通常使用通用网关接口 CGI 产生动态网页)。

(4) Web 浏览器收到 Web 响应，并加载，解析后显示在用户的屏幕上。

上述过程如图 8-1 所示。

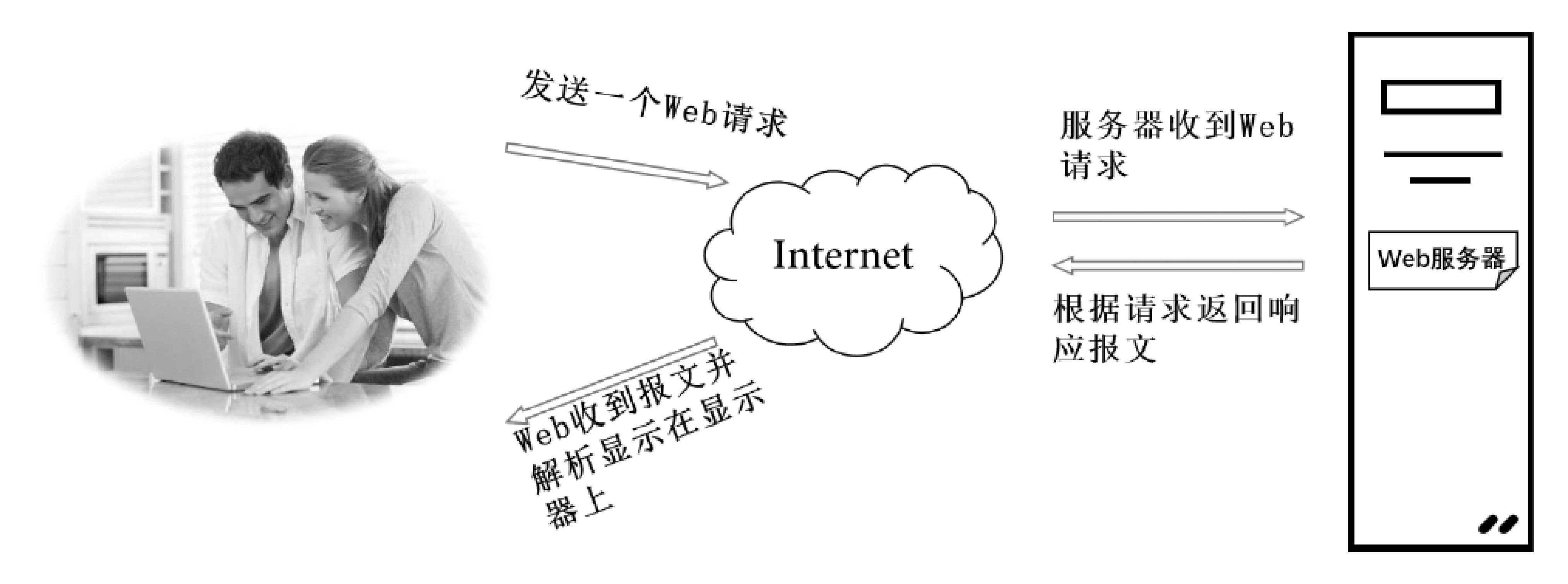


图 8-1　Web 应用在服务器上的工作过程

8.1.2　为 Web 应用创建一个 UI

本章将以一个“网上书店”为例，来对 Web 应用开发进行简要介绍。

首先开始编写网上书店的视图代码，这会创建 Web 应用的用户界面(User Interface，UI)。在 Web 开发中，用户界面通常使用 HTML 技术来创建。如果读者对 HTML 技术还不是很了解，可能需要花些时间来掌握它。可以在网上参考一些相关资料，这样可以快速地熟悉 HTML 的一些基本用法，也可以阅读一些相关的 HTML 教材去深入了解这门非常重要的 Web 开发技术。由于本书篇幅所限，本章拟不介绍 HTML 技术。

根据平时浏览网站的习惯，在输入网址后，首先进入的应该是网上书店的首页。它一般包括一个欢迎标题、一个书店的图标，一个超链接子页面和相关的文字介绍。HTML 文件代码如下。

【例 8-1】index.html 文件的内容：

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
```

```
    <title>BookStore</title>
</head>
<body>
    <h1>Welcome to My Book Store.</h1>
    <img src="imgs/books.png">
    <h3>
    please choose your favorite book, click <a href="...">here</a>.
    </h3>
    <p>
    <strong> Enjoy!</strong>
    </p>
</body>
</html>
```

将上述内容保存为.html 文件后，用浏览器打开，效果如图 8-2 所示。

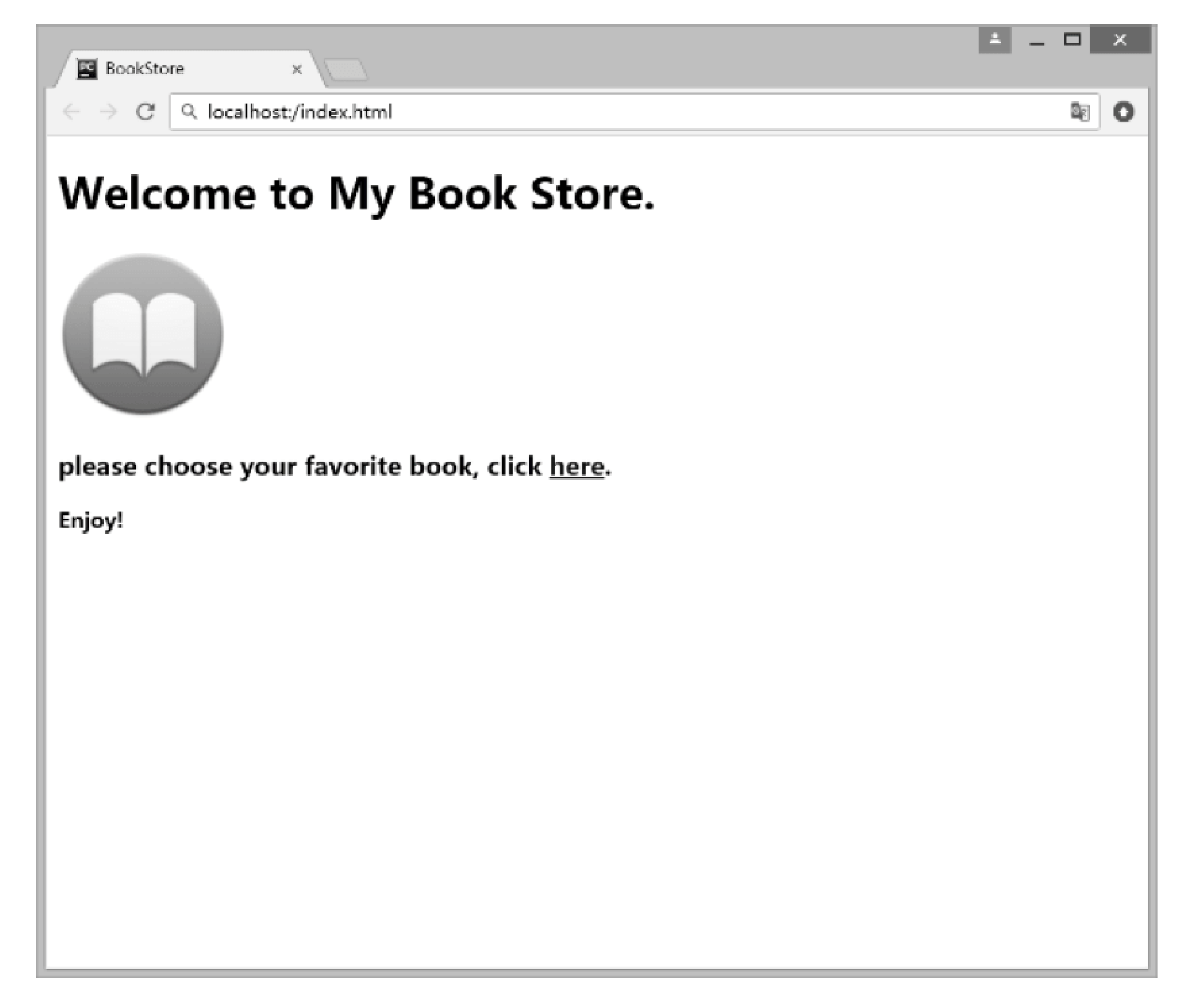


图 8-2　例 8-1 的 HTML 文件使用浏览器打开后的效果

在实际的网站开发中，为每个网页都编写一个 HTML 文件是极为低效的。这时可以通过编写模板文件来快速生成 Web 页面。例如，可以编写一个模板文件并保存为 yate.py，之后，通过替换 yate.py 中的内容实现网页的快速生成。

【例 8-2】模板文件 yate.py：

```
def start form(the url, form type="POST"):
    return('<form action="' + the url + '" method="' + form type + '">')

def end_form(submit_msg="Submit"):
```

```
    return('<input type=submit value="' + submit msg + '"></form>')

def radio button(rb name, rb value):
    return('<input type="radio" name="' + rb name +
                 '" value="' + rb value + '"> ' + rb value + '<br />')

def u list(items):
    u string = '<ul>'
    for item in items:
        u string += '<li>' + item + '</li>'
    u string += '</ul>'
    return(u string)

def header(header text, header level=2):
    return('<h' + str(header level) + '>' + header text +
          '</h' + str(header level) + '>')
def para(para text):
    return('<p>' + para text + '</p>')

def link(the link,value):
    link string = '<a href="' + the link + '">' + value + '</a>'
    return(link_string)
```

在以后的应用中，只需要用 import yate 命令加载该文件，通过语句：

```
print(yate.header('Book Detail:'))
```

就可以快速地定义 Web 页面标题，这显然是非常方便的。

8.2　使用 MVC 设计 Web 应用

在前面的例子中，已经建立了两个代码文件，并且在例 8-1 中还引入了一个图片文件。那么应如何保存这些文件？何种方式最佳呢？

一个最受大家认可的 Web 应用应该遵循 MVC(Model View Controller)模式，这种模式更加有利于维护和管理各个功能模块和组件。

MVC 是“模型(Model)—视图(View)—控制器(Controller)”的缩写，它是一种软件设计典范，用一种将业务逻辑、数据、界面显示三者分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

模型：定义数据库相关的内容，一般放在 models.py 文件中。

视图：定义 HTML 等静态网页文件相关的内容，即 HTML、CSS、JS 等前端文件。

控制器：定义业务逻辑相关的内容，也就是我们运行的主要代码。

通俗来说，MVC 设计就是一种文件的组织和管理形式，它将不同的文件分类存放，以方便后期的管理。采用 MVC 设计的优点是很明显的，分层的设计降低了 Web 应用的耦合性，允许更改视图层代码而不用重新编译模型和控制器代码。因为多个视图可以共用一个模板来实现，所以 MVC 方法具有可重用性高、部署快、可维护性好等特点。

图 8-3 就是推荐的一种文件存储结构。

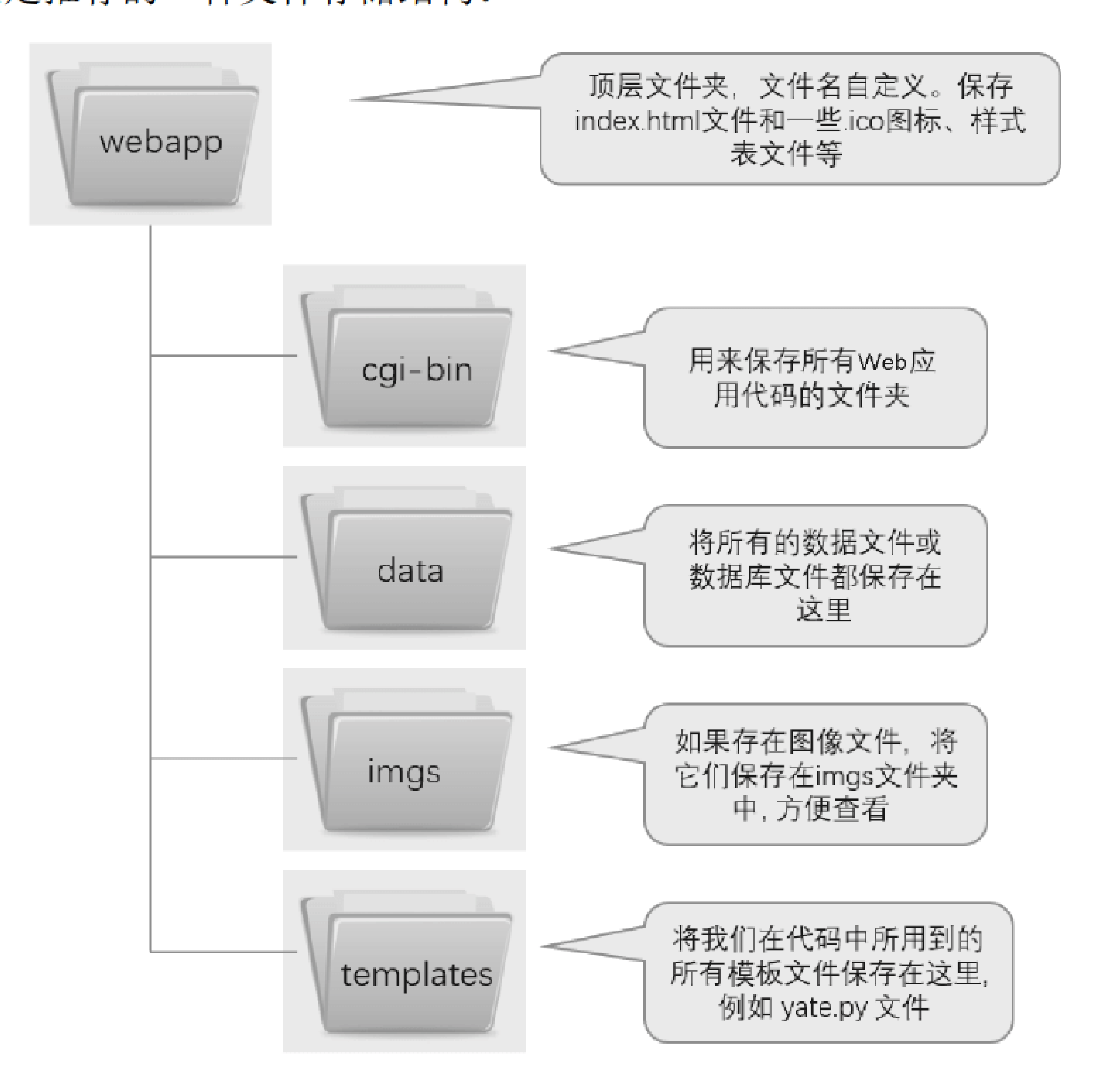


图 8-3　MVC 式文件存储

8.3　使用 CGI 将程序运行在服务器上

通用网关接口(Common Gateway Interface，CGI)是外部应用程序(CGI 程序)与 Web 服务器之间的接口标准，允许 Web 服务器运行一个服务器端程序，简称 CGI 脚本。

一般情况下，CGI 脚本都保存在 cgi-bin 文件夹中，这样，Web 服务器能方便地找到 CGI 脚本。所有的 Web 应用都要在 Web 服务器上运行，实际上，所有的 Web 服务器都支持 CGI，无论是 Apache、IIS、Nginx、Lighttpd 还是其他服务器，它们都支持用 Python 编写的 CGI 脚本。这些 Web 服务器功能都十分强大，当然在使用过程中也较为复杂。相对于这里较小的项目规模，我们还可以使用 Python 自带的简单的 Web 服务器，这个 Web 服务器包含在 http.server 库模块中。

运行下面的程序，就可以启动一个支持 CGI 的 Web 服务器。

【例 8-3】启动 Web 服务器：

```
from http.server import HTTPServer, CGIHTTPRequestHandler
port = 8080
httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
print("Starting simple httpd on port: " + str(httpd.server port))
httpd.serve_forever()
```

CGI 标准指出，服务器端程序(CGI 脚本)生成的任何输出都将会由 Web 服务器捕获，

并发送到等待的 Web 浏览器。具体来说，它会捕获发送到 stdout(标准输出)的所有内容。

一个 CGI 脚本由两部分组成，第一部分输出 Response Headers，第二部分输出常规的 HTML 文件。其中 Response Headers 部分一般用如下语句来实现：

```
print("Content-type:text/html\n")
```

例如在“网上书店”的例子中，index.html 页面的超链接“here”会链接到图书清单页面。该页面包含一个标题、一个通过服务器应用动态生成的图书列表和返回按钮。可以使用 yate 模板来实现该 CGI 脚本。

【例 8-4】图书清单页面。

① book_list_view.py：

```
import cgitb
cgitb.enable()
#启用这个模块时，会在 Web 浏览器上显示详细的错误信息。enable()函数打开 CGI 跟踪
#CGI 脚本产生一个异常时，Python 会将消息显示在 stderr(标准出错文件)上。CGI 机制会忽略
这个输出，因为它想要的只是 CGI 的标准输出(stdout)
from templates import yate
import book service

print("Content-type:text/html\n")#Response Headers
#网页内容：由 HTML 标签组成的文本
print('<html>')
print('<head>')
print('<title>Book List</title>')
print('</head>')
print('<body>')
print('<h2>Book List:</h2>')
print(yate.start form('book detail view.py'))  #调用模板生成图书列表
book dict=book service.get book dict()  #调用图书管理模块提取图书信息
for book name in book dict:
   print(yate.radio button('bookname',book dict[book name].name))
print(yate.end form('detail'))
print(yate.link("/index.html",'Home'))
print('</body>')
print('</html>')
```

② book_service.py：

```
from templates import Book

def get book dict():
   book dict={}
   try:
      with open('data\\book.txt','r') as book file:
         for each line in book file:
            book=parse(each line)
            book dict[book.name]=book
   except IOError as ioerr:
      print("IOErr:",ioerr)
```

```
    return(book_dict)

def parse(book_info):
    (name,author,price)=book_info.split(';')
    book=Book.Book(name,author,price)
    return(book)
```

③　Book.py：

```
from templates import yate

class Book:
    def __init__(self,name,author,price):
        self.name=name
        self.author=author
        self.price=price

    @property
    def get_html(self):
        html_str=''
        html_str+=yate.header('BookName:',4)+yate.para(self.name)
        html_str+=yate.header('Author:',4)+yate.para(self.author)
        html_str+=yate.header('Price:',4)+yate.para(self.price)
        return(html_str)
```

上面的代码一共包含三个文件，图书列表视图文件 book_list_view.py、图书处理逻辑文件 book_service.py 和定义书籍类的文件 Book.py。其中，book_service.py 文件从本地的 book.txt 文件中提取出书籍信息，调用 Book 类生成 Book 对象，并以字典格式保存所有的书籍信息后，返回给 book_list_view.py 中的 book_dict，然后通过模板生成书籍列表内容。

待显示的图书信息以“书名；作者；价格”的格式保存在 book.txt 文件中，如图 8-4 所示。

```
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
The Linux Programming Interface: A Linux and UNIX System Prog;Michael Kerrisk;
$123.01
HTML5 and CSS3, Illustrated Complete (Illustrated Series);Jonathan Meersman Sasha
Vodnik;$32.23
Understanding the Linux Kernel;Daniel P. Bovet Marco Cesati;$45.88
Getting Real;Jason Fried, Heinemeier David Hansson, Matthew Linderman;$87.99
```

图 8-4　待显示的图书信息

运行 book_list_view.py 文件，可以得到如图 8-5 所示的输出。

最后一个页面是详细的图书信息页面。在这个 CGI 脚本中，首先要获得上级表单提交给 Web 服务器的值 BookName，并通过 BookName 判断需要动态加载的书籍信息。完成上面的目标可以通过使用 CGI 库的 cgi.FieldStorage()来实现。

cgi.FieldStorage()方法访问 Web 浏览器发送给 Web 服务器的数据，并将这些数据保存为一个 Python 字典。在确定书籍名称后，即可访问本地 book 数据字典，之后调用 yate 模板生成视图页面。

```
Content-type:text/html

<html>
<head>
<title>Book List</title>
</head>
<body>
<h2>Book List:</h2>
<form action="book_detail_view.py" method="POST">
<input type="radio" name="bookname" value="HTML5 and CSS3, Illustrated Complete (Illustrated Series)"> HTML5 and CSS3
<input type="radio" name="bookname" value="Getting Real"> Getting Real<br />
<input type="radio" name="bookname" value="Understanding the Linux Kernel"> Understanding the Linux Kernel<br />
<input type="radio" name="bookname" value="The Linux Programming Interface: A Linux and UNIX System Prog"> The Linux
<input type=submit value="detail"></form>
<a href="/index.html">Home</a>
</body>
</html>
```

图 8-5　book_list_view.py 的运行效果

这里需要注意的是，表单可能返回的是一个 None 值，即用户可能没有选取任何书籍。所以需要用一个 try/except 语句来保护代码，当 bookname 值为空时，提醒用户选择书籍。具体的代码包含在下面的例子中。

【例 8-5】图书详细页面。

book_detail_view.py：

```
import cgitb
cgitb.enable()

import cgi
import templates.yate as yate
import book_service

form_data = cgi.FieldStorage()

print("Content-type:text/html\n")
print('<html>')
print('<head>')
print('<title>Book List</title>')
print('</head>')
print('<body>')
print(yate.header('Book Detail:'))
try:
   book_name = form_data['bookname'].value
   book_dict=book_service.get_book_dict()
   book=book_dict[book_name]
   print(book.get_html)
except KeyError as kerr:
   print(yate.para('please choose a book...'))
print(yate.link("/index.html",'Home'))
print(yate.link("/cgi-bin/book_list_view.py",'Book List'))
print('</body>')
print('</html>')
```

所有文件保存后，整个工程文件的结构如图 8-6 所示。

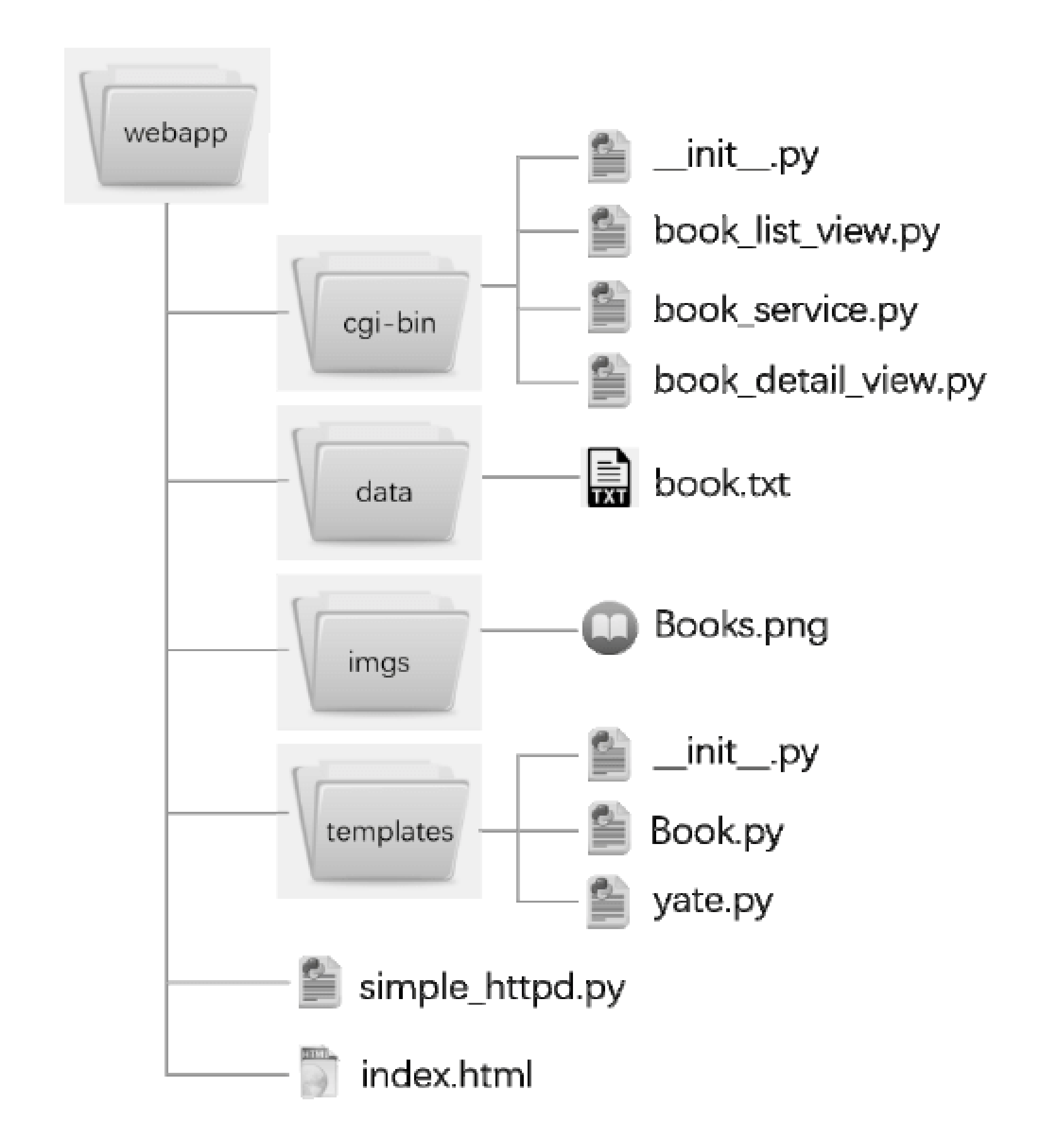


图 8-6　全部工程文件的结构

首先运行服务器文件 simple_httpd.py，得到输出“Starting simple_httpd on port: 8080”后，说明服务器已经开始正常运行并监听输出内容。

这时在浏览器输入 http://127.0.0.1:8080/(本地服务器地址)，就可以打开我们的“网上书店”页面，运行效果如图 8-7 所示(请读者注意图 8-2 与图 8-7 的区别)。

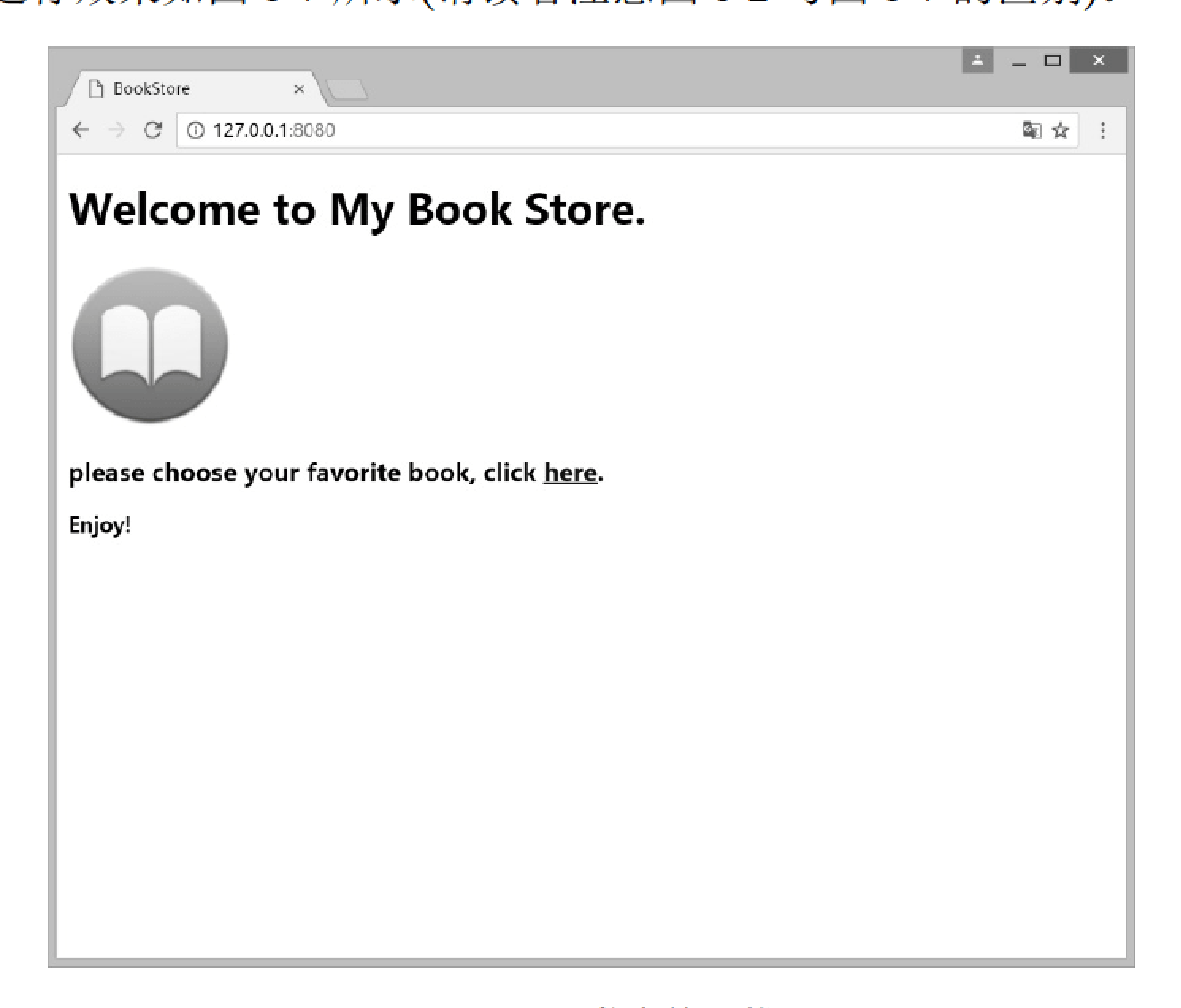


图 8-7　网上书店首页截图

单击 here 链接后，执行 book_list_view.py 文件，运行效果如图 8-8 所示。

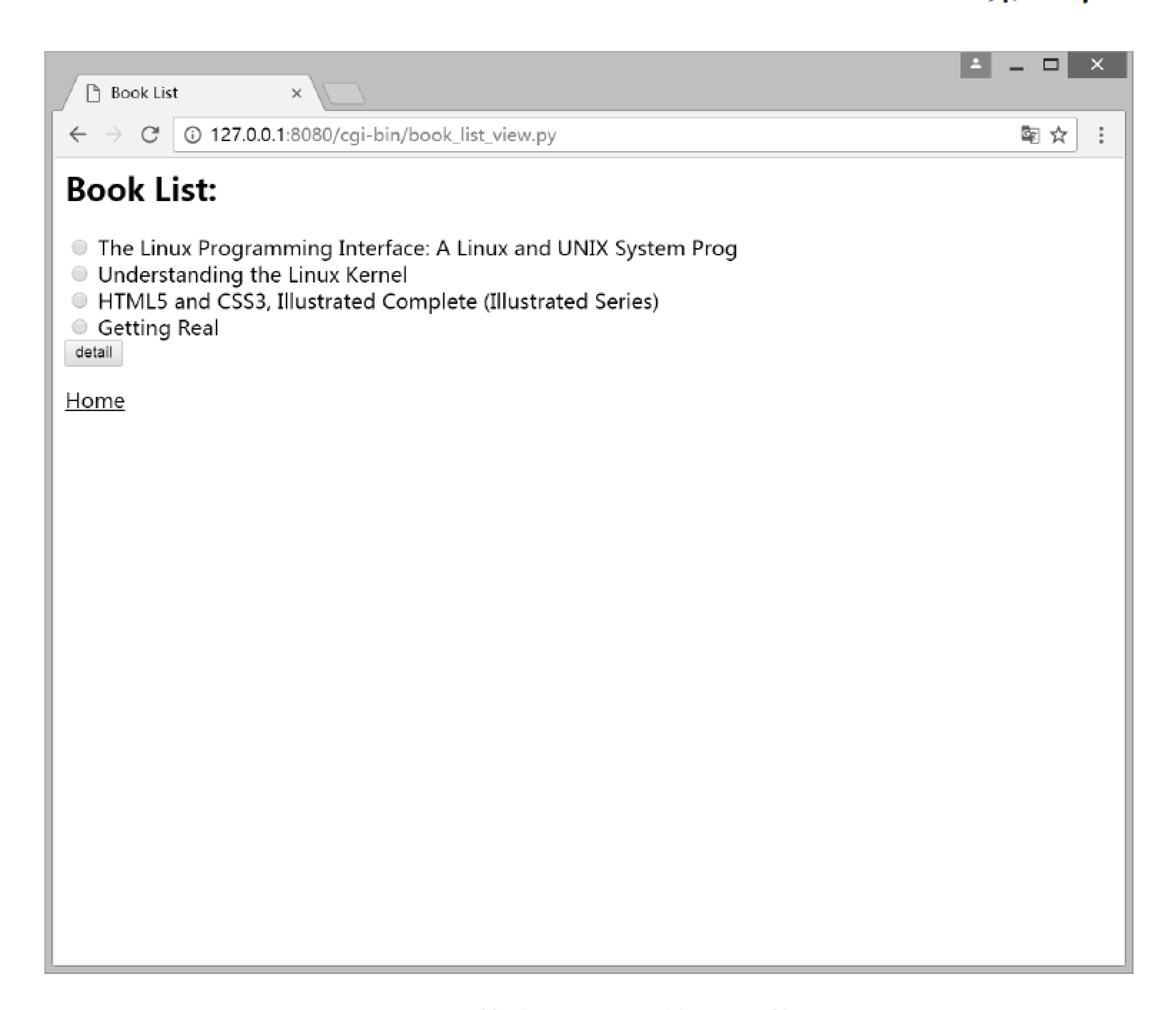


图 8-8　单击 here 后的页面截图

选择书籍后，单击 detail，可以查看图书细节，这时跳转到 book_detail_view.py 文件，运行效果如图 8-9 所示。

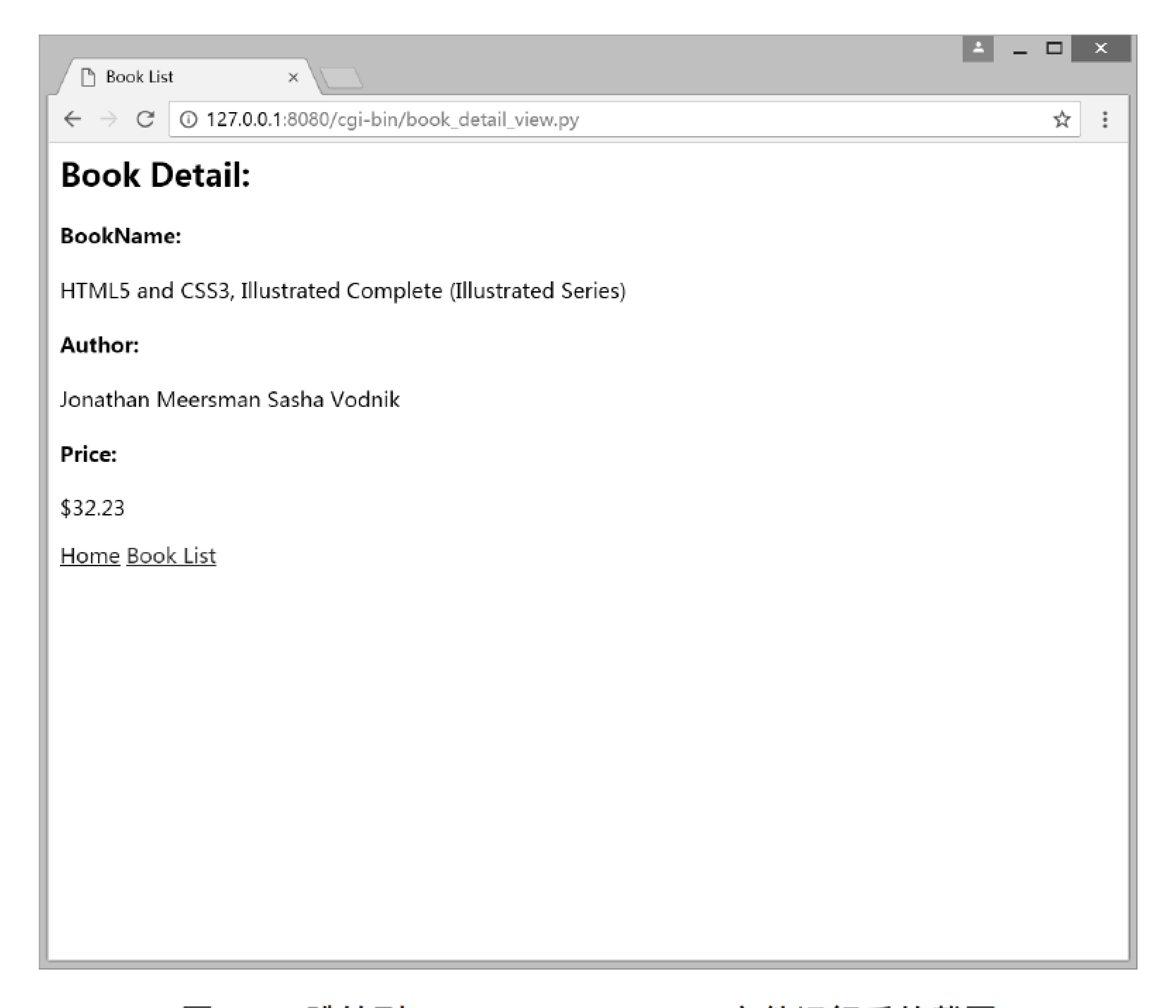


图 8-9　跳转到 book_detail_view.py 文件运行后的截图

控制台监控到的请求信息如图 8-10 所示。

```
D:\Anaconda3\python.exe D:/web开发/simple_httpd.py
Starting simple_httpd on port: 8080
127.0.0.1 - - [16/Mar/2017 09:01:25] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2017 09:01:25] "GET /imgs/books.png HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2017 09:01:25] code 404, message File not found
127.0.0.1 - - [16/Mar/2017 09:01:25] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [16/Mar/2017 09:03:26] "GET /cgi-bin/book_list_view.py HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2017 09:03:26] command: D:\Anaconda3\python.exe -u D:\web开发\cgi-bin\book_list_view.py ""
127.0.0.1 - - [16/Mar/2017 09:03:27] CGI script exited OK
127.0.0.1 - - [16/Mar/2017 09:07:54] "POST /cgi-bin/book_detail_view.py HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2017 09:07:54] command: D:\Anaconda3\python.exe -u D:\web开发\cgi-bin\book_detail_view.py ""
127.0.0.1 - - [16/Mar/2017 09:07:54] CGI script exited OK
127.0.0.1 - - [16/Mar/2017 09:08:31] "GET /cgi-bin/book_list_view.py HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2017 09:08:31] command: D:\Anaconda3\python.exe -u D:\web开发\cgi-bin\book_list_view.py ""
127.0.0.1 - - [16/Mar/2017 09:08:31] CGI script exited OK
127.0.0.1 - - [16/Mar/2017 09:09:02] "POST /cgi-bin/book_detail_view.py HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2017 09:09:02] command: D:\Anaconda3\python.exe -u D:\web开发\cgi-bin\book_detail_view.py ""
127.0.0.1 - - [16/Mar/2017 09:09:03] CGI script exited OK
```

图 8-10　控制台消息

8.4　案例实训：Web 页面获取表格内容并显示

前面以一个“网上书店”实例，对如何使用 Python 进行 Web 开发做了详细的讲解。本案例以获取表格内容并显示为例，再次对整个 Web 开发的主要过程进行回顾。在本例中，首先由用户通过一个 HTML 页面输入数据，并提交给 CGI 服务器，然后通过 CGI 脚本获取其内容并返回到 HTML 页面中显示出来。

首先，建立一个工程文件夹 www，并在文件夹下建立 simple_httpd.py 文件。在该文件中输入下面的代码并运行，打开 CGI 服务：

```
from http.server import HTTPServer, CGIHTTPRequestHandler
port = 8080
httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
print("Starting simple httpd on port: " + str(httpd.server port))
httpd.serve_forever()
```

其次，在 www 文件夹中建立一个首页文件 index.html，代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Web 开发案例</title>
</head>
</head>
<body>
    <h1>Web 开发案例</h1>
    <div>
        <form name="input" action="/cgi-bin/test1.py" method="get">
            姓名: <input type="text" name="name"><br>
            学号: <input type="text" name="id"><br>
```

```
            专业: <input type="text" name="specialty"><br>
            分数: <input type="text" name="score"><br>
            <input type="submit" value="提交">
        </form>
    </div>
</body>
</html>
```

在浏览器的地址栏中输入地址 http://127. 0. 0. 1:8080/(本地服务器地址)，即可访问到 index. html 文件，效果如图 8-11 所示。

注意： 在 index.html 文件的 form 表单里设置了语句 action="/cgi-bin/test1.py"，旨在实现单击页面中的“提交”按钮后能够调用同一目录下 cgi-bin 文件夹中的 test1.py 文件。



图 8-11　在浏览器中输入 http://127.0.0.1:8080/后显示的页面

最后，在 www 文件夹下新建文件夹 cgi-bin，用来保存我们的 test1.py 文件。

test1.py 文件的代码如下：

```
import cgi

form = cgi.FieldStorage()
print ("Content-Type: text/html")
print ("")
print ("<html>")
print ("<h2>CGI 服务器接收到的数据</h2>")
print ("<p>")
print ("用户提交的信息:<br>")
print ("<b>姓名:</b> " + form["name"].value + "<br>")
print ("<b>学号:</b> " + form["id"].value + "<br>")
print ("<b>专业:</b> " + form["specialty"].value + "<br>")
print ("<b>分数:</b> " + form["score"].value + "<br>")
```

```
print ("</p>")
print ("</html>")
```

如前所述，index.html 页面中提交的表单内容会被封装在一个 cgi.FieldStorage 对象中。通过 form = cgi.FieldStorage()语句即可在 CGI 服务器中获取其内容，然后通过 print 语句将 form 中的内容输出，并返回给浏览器。

创建完成后，我们测试一下实际运行的效果。在 Web 页面中输入图 8-12 所示的数据后单击“提交”按钮，提交后的页面如图 8-13 所示。可见，Web 表格中的内容被提取出来了。



图 8-12　在表单中输入数据



图 8-13　单击“提交”按钮后显示的内容

本 章 小 结

本章先是通过一个“网上书店”例子初步介绍了使用 Python 语言进行 Web 开发的过程，然后以提取 Web 表格内容为案例，回顾了 Web 开发的主要过程。学习过其他 Web 开发语言的读者可能会感觉到，相对于 Java、C#等语言，使用 Python 开发 Web 是一件非常轻松的事情。没有复杂的步骤，也没有冗长的代码，而且 Python 语言可以很好地与其他语言结合，这使得它在页面交互方面的应用更加轻松。因此，很多大型的网站，如知乎、网易、腾讯、搜狐等，都在网站开发中使用了大量的 Python 技术。

习　　题

1. 填空题

(1) 基于 Web 开发的应用具有(　　　)、(　　　)、(　　　)、(　　　)、(　　　)等优点。

(2) MVC 模式是(　　　)、(　　　)、(　　　)的缩写。

(3) 在实际的网站开发中，为每个网页都编写一个 HTML 文件是极为低效的。这时可以通过编写(　　　)来快速生成 Web 页面。

2. 选择题

(1) 更具体地说，MVC 是一种(　　)。

A. 文件组织和管理形式　　B. 软件设计方法

C. 界面设计方法　　D. Web 开发方法

(2) 在 Web 开发中，CGI 指的是(　　)。

A. 计算机图形接口(Computer Graphics Interface)

B. 全球小区识别码(Cell Global Identifier)

C. 计算机生成影像(Computer-Generated Imagery)

D. 通用网关接口(Common Gateway Interface)

3. 简答题

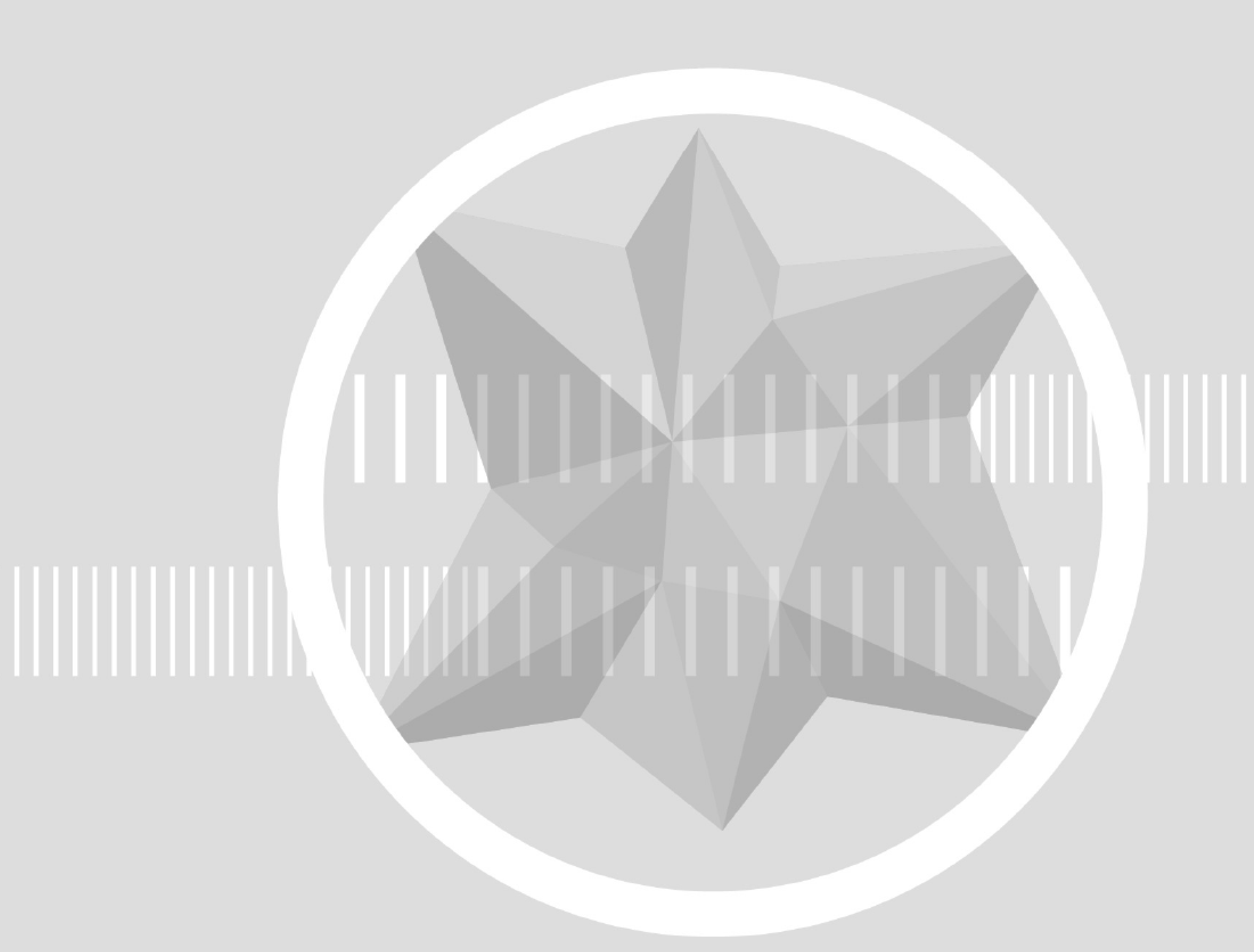
(1) MVC 设计模式将应用程序按用户界面的功能划分为哪几类？

(2) 使用 MVC 模式有什么好处？

(3) 什么是 CGI？

4. 实验操作题

如何使用 Python 自带的 http.server 库模块建立服务器？请写出具体代码。



第 9 章

使用 Python 进行数据分析

本章要点

(1) 使用 Python 进行数据挖掘的原因。
(2) NumPy 库。
(3) SciPy 库。
(4) Matplotlib 库。
(5) Pandas 库。
(6) 数据可视化。

学习目标

(1) 了解 Python 在数据挖掘方面的应用。
(2) 掌握 NumPy 库的安装和使用。
(3) 掌握 SciPy 库的安装和使用。
(4) 掌握如何使用 Matplotlib 库画图。
(5) 掌握 Pandas 库的两种重要类型 DataFrame、Series。
(6) 掌握利用 Python 进行数据可视化的方法。

Python 语言在进行数据分析方面有着强大的优势，它处理速度快，编程效率高，支持数据的可视化，这些特性使其很快成为进行数据分析的主流编程语言之一。本章将对其在数据分析上的应用和优势进行详细介绍。

由于数据挖掘是数据分析的高级形式，所以本章内容较多地涉及数据挖掘。

9.1 数据挖掘简介

数据挖掘是从大量的数据(包括文本)中挖掘出隐含的、先前未知的、对决策有潜在价值的关系、模式和趋势，并用这些知识和规则建立用于决策支持的模型，提供预测性决策支持的方法、工具和过程。

数据挖掘有助于企业发现业务的趋势，揭示已知的事实，预测未知的结果，因此数据挖掘已成为企业保持竞争力的必要手段。

数据挖掘由以下步骤组成。

(1) 数据清理(消除噪声和删除不一致的数据)。
(2) 数据集成(将多种数据源组合在一起)。
(3) 数据选择(从数据库中提取与分析任务相关的数据)。
(4) 数据变换(通过汇总或聚集操作，把数据变换和统一成适合挖掘的形式)。
(5) 数据挖掘(基本步骤，使用智能的方法提取数据模式)。
(6) 模式评估(根据某种兴趣度度量，识别代表知识的真正有趣的模式)。
(7) 知识表示(使用可视化和知识表示技术，向用户提供挖掘的知识)。

其中，步骤(1) ~ (4)是数据预处理的不同形式，为数据挖掘做准备。

9.2　为什么选择 Python 进行数据挖掘

Python 本身是一门简单易学的语言，它优雅的语法和动态的类型，结合它的解释性，使其在很多领域成为编写脚本或开发程序的理想语言。相对于 Matlab 开发工具，Python 可以完成 Matlab 可以完成的所有任务。而且在大多数情况下，相同功能的 Python 代码会比 Matlab 代码更加简洁和易懂。

同时，Python 毕竟还是一门编程语言，它在开发网页、开发游戏、编写网络爬虫获取数据等方面的应用也是 Matlab 所不能及的。

Python 语言以高效和简洁著称，致力于用最简洁、最简短的代码完成任务。相对于 Java、C/C++等语言，Python 快速编程、快速验证的特性又十分适合数据挖掘所要求的时效性。

Python 语言经常受人诟病的是它的运行效率，但是，Python 还被称为胶水语言，它可以很好地兼容 C/C++语言，核心部分用 C/C++等更高效的语言来编写，然后通过 Python 粘合，因此其效率问题可以很好地得到解决。这一点早在第 1 章已经提到。

同时，随着大量应用于数据挖掘的程序库的开发，例如 NumPy、SciPy、Matplotlib 和 Pandas 等，在大多数的数据科学计算任务上，Python 语言的运行效率已经可以媲美 C/C++语言了。

随着近年来 Python 被越来越多的人所关注，应用于数据处理的 Python 程序库与日俱增，可以预见的是，Python 语言正在慢慢地成为数据科学领域的主流编程语言。大数据时代的到来，更使 Python 有了用武之地。

9.3　Python 的主要数据分析工具

Python 语言本身的数据处理能力并不是很强，它主要依靠众多的第三方库来增强其能力，像常用的 NumPy 库、SciPy 库、Matplotlib 库、Pandas 库、Scikit-Learn 库、Keras 库和 Gensim 库等。本节主要就这些库的基本应用进行介绍。

常规版本的 Python 需要在安装完成后另外下载相应的第三方库来安装上述库文件，而如果安装的是 Anaconda 发行版本的 Python，那么它可能已经同时安装了下面的库：NumPy、SciPy、Matplotlib、Pandas、Scikit-Learn。

Anaconda 是一个专门用于科学计算的 Python 版本，里面包含了大量关于数据计算和数据挖掘的工具。如果用户应用 Python 专门从事大量关于数据科学计算的工作，Anaconda 版本的 Python 将不需要我们再一个一个地安装各种库。

9.3.1　NumPy 库

Python 本身并没有提供数组功能。虽然其列表功能已经可以完全代替数组的功能，但在进行数据科学计算时，常常需要面对大量的数据和复杂的运算过程。这时，列表就不能很好地适应我们的需求了。NumPy 是专门为了进行严格的数据处理而开发的，它提供了一

个非常强大的 N 维数组对象 Array 和实用的线性代数、傅里叶变换和随机数生成函数，这个工具可以用来存储和处理大型的矩阵，且处理速度是 C 语言级别的！同时，后面将要介绍的 SciPy、Matplotlib、Pandas 等库都依赖于它。

1. 安装 NumPy 库

在 Windows 系统中，NumPy 可以像安装其他库一样，通过 pip 安装：

```
pip install NumPy
```

也可以先自行下载所需要的其他版本的 NumPy，然后用如下命令来安装：

```
python setup.py install
```

在 Linux 系统中也可以通过上面的方法进行安装。此外，还可以通过 Linux 自带的软件管理器进行安装，如在 Ubuntu 版本下，可以通过使用如下命令进行安装：

```
sudo apt-get install Python-NumPy
```

安装完成后，可以通过下面的语句进行测试，看安装是否成功：

```
import numpy as np
print(np.version.version)
```

如果安装正确，将会输出 NumPy 的版本号。

2. 多维数组 ndarray

ndarray 是 NumPy 最重要的组成部分，该对象是一个快速而又灵活的大数据集容器，可以利用这种数组对整块的数据执行数学运算。

(1) 创建 ndarray。

创建数组最简单的办法就是使用 array 函数。它接收一切序列类型的对象(例如 list、tuple、其他数组等)，然后生成一个含有传入数据的 NumPy 数组。

【例 9-1】生成 ndarray 对象：

```
import numpy as np

data1 = [6, 7.5, 4, 58, 1]     #生成一个列表对象
data2 = (5, 9, 6.3, 7, 0, 1)  #生成一个元组对象

arr1 = np.array(data1)  #利用列表对象生成 ndarray 对象
arr2 = np.array(data2)  #利用元组对象生成 ndarray 对象

print(type(arr1))  #输出 arr1 的数据类型
print(arr1)
print(arr2)
```

输出结果：

```
<class 'numpy.ndarray'>
[  6.    7.5   4.   58.    1. ]
[ 5.   9.   6.3  7.   0.   1. ]
>>> |
```

利用嵌套序列(例如由一组等长的列表组成的列表)可以生成一个多维数组。

【例 9-2】生成多维数组：

```
import numpy as np

data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  #一个由列表组成的列表
arr = np.array(data)  #转换为一个二维数组

print(arr)
print("数组维数: " + str(arr.ndim))  #输出数组的维度
print("数组类型: " + str(arr.shape))  #输出数组的行列数
```

输出结果为：

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
数组维数: 2
数组类型: (3, 3)
>>>
```

在生成数组的时候，还可以指定其数据类型，例如 numpy.int32、numpy.int16 和 numpy.float64 等。

【例 9-3】生成数组时指定数据类型：

```
import numpy as np

data = [1.77, 2, 3, 4, 5, 6]  #生成一个列表对象
arr = np.array(data,np.int32)  #转换为一个数组对象
print(arr)
```

输出结果：

```
[1 2 3 4 5 6]
>>>
```

可见，输出的列表已经自动保留整数部分。

(2)　创建特殊数组。

除了上面提到的可以利用 array 创建数组外，还可以利用 numpy.zeros、numpy.ones、numpy.eye 等方法构造特定的数组。

【例 9-4】创建特殊数组：

```
import numpy as np

arr1 = np.zeros((3, 4))  #生成一个 3 行 4 列的全 0 数组
arr2 = np.ones((3, 4))  #生成一个 3 行 4 列的全 1 数组
arr3 = np.eye(3)  #生成一个三阶单位数组

print("全 0 数组: \n", arr1)
print("全 1 数组: \n", arr2)
print("单位数组: \n", arr3)
```

输出结果：

```
全0数组：
 [[ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]]
全1数组：
 [[ 1.   1.   1.   1.]
 [ 1.   1.   1.   1.]
 [ 1.   1.   1.   1.]]
单位数组：
 [[ 1.   0.   0.]
 [ 0.   1.   0.]
 [ 0.   0.   1.]]
>>>
```

注意： 在第 3 章中，我们曾使用双重循环来构造一个 3×3 的全 0 矩阵(参见例 3-29)。这里，仅使用一条语句“arr0 = np.zeros((3, 3))”即可实现同样的功能。由此可见，恰当地运用第三方扩展库进行 Python 编程是何等的简洁！其实，不仅仅是程序简洁，执行效率也会有 1~2 个数量级的提高(C/C++的执行效率)。

表 9-1 中列出了常用的一些数组创建函数。另外，由于 NumPy 库主要用于数据科学计算，所以在默认的情况下，数据类型基本上都是 float64(浮点类型)。

表 9-1 常用的数组创建函数

函 数 名	说 明
array	将输入的数据(列表、元组、数组或其他序列类型)转换成 ndarray
asarray	将输入转换为 ndarray，如果输入一个 ndarray 则不会进行复制
arange	类似于 Python 内置的 range()函数，但返回的是一个 ndarray 类型，而不是 list 类型
ones、ones_like	根据指定的形状和格式(dtype)创建一个全为 1 的数组。ones_like 以另一个数组做参数生成一个全为 1 的数组
zeros、zeros_like	功能类似于 ones、ones_like，不过生成的是全为 0 的数组
empty、empty_like	生成一个新的数组但只分配存储空间，而随机生成一些未初始化的值
eye、identity	创建一个正方形的对角线全为 1 的单位数组

9.3.2 SciPy 库

NumPy 库的加入，使人们可以高效地处理数据了，但是，尽管 NumPy 提供了多维数组的功能和大量的生成函数，但它并不是真正意义上的矩阵。例如，当两个数组相乘时，只是对应元素的相乘，而非数学意义上的矩阵乘法。SciPy 库则提供了真正的矩阵，以及大量的基于矩阵运算的对象和函数。

SciPy 包含的功能有最优化、线性代数、积分、插值、拟合、特殊函数、快速傅里叶变换、信号处理和图像处理、常微分方程求解和其他科学与工程中常用的计算。这些功能在进行数据挖掘时都是必不可少的。

SciPy 库依赖于 NumPy 库，因此在安装 SciPy 库之前，须先安装好 NumPy 库，利用 SciPy+NumPy 组合可以解决大量的原本需要 C++或者 Matlab 才能解决的问题。

1. 安装 SciPy 库

SciPy 库的安装过程类似于 NumPy 库，同样可以使用 pip 安装或者自行安装，在

Ubuntu 环境下也可以通过如下命令进行安装：

```
sudo apt-get install Python-SciPy
```

安装成功后，可以用如下命令测试是否安装成功：

```
import SciPy, numpy
print(SciPy.version.full_version)
```

如果安装成功，将会显示 SciPy 的版本号和 True 值。

2. 常用 SciPy 工具包

表 9-2 是一些常用的 SciPy 工具包。因本书篇幅所限，此处不一一详述，具体内容可参考 http://www.SciPy.org 网站，或参考 *SciPy and NumPy* 一书(参考文献[21])。

表 9-2　常用 SciPy 工具包

SciPy 工具包	功　能
cluster	层次聚类(cluster.hierarchy) 矢量量化/K 均值(cluster.vq)
constants	物理和数学常量 转换方法
fftpack	离散傅里叶变换算法
integrate	积分例程
interpolate	插值(线性的、三次方的等)
io	数据输入和输出
linalg	采用优化 BLAS 和 LAPACK 库的线性代数函数
maxentropy	最大熵模型的函数
ndimage	n 维图像工具包
odr	正交距离回归
optimize	最优化(寻找极小值和方程的根)
signal	信号处理
sparse	稀疏矩阵
spatial	空间数据结构和算法
special	特殊数学函数，如贝塞尔函数(Bessel)或雅可比函数(Jacobian)
stats	统计学工具包

【例 9-5】使用 SciPy 求解下面的线性方程组：

$$\begin{cases} 2w + x - 5y + z = 8 \\ w - 3x - 6z = 9 \\ 2x - y + 2z = -5 \\ w + 4x - 7y + 6z = 0 \end{cases}$$

程序如下：

```
import scipy
from scipy import linalg

a= scipy.mat('[2 1 -5 1;1 -3 0 -6;0 2 -1 2;1 4 -7 6]')
b=scipy.mat('[8;9;-5;0]')
solve = linalg.solve(a, b)

print(solve)
```

运行结果为：

```
[[ 3.]
 [-4.]
 [-1.]
 [ 1.]]
>>>
```

即方程组的解为：

$$\begin{cases} w = 3 \\ x = -4 \\ y = -1 \\ z = 1 \end{cases}$$

【**例 9-6**】使用 SciPy.ndimage 对图像进行处理：

```
from scipy import ndimage
from scipy import misc
import pylab as pl
ascent = misc.ascent()

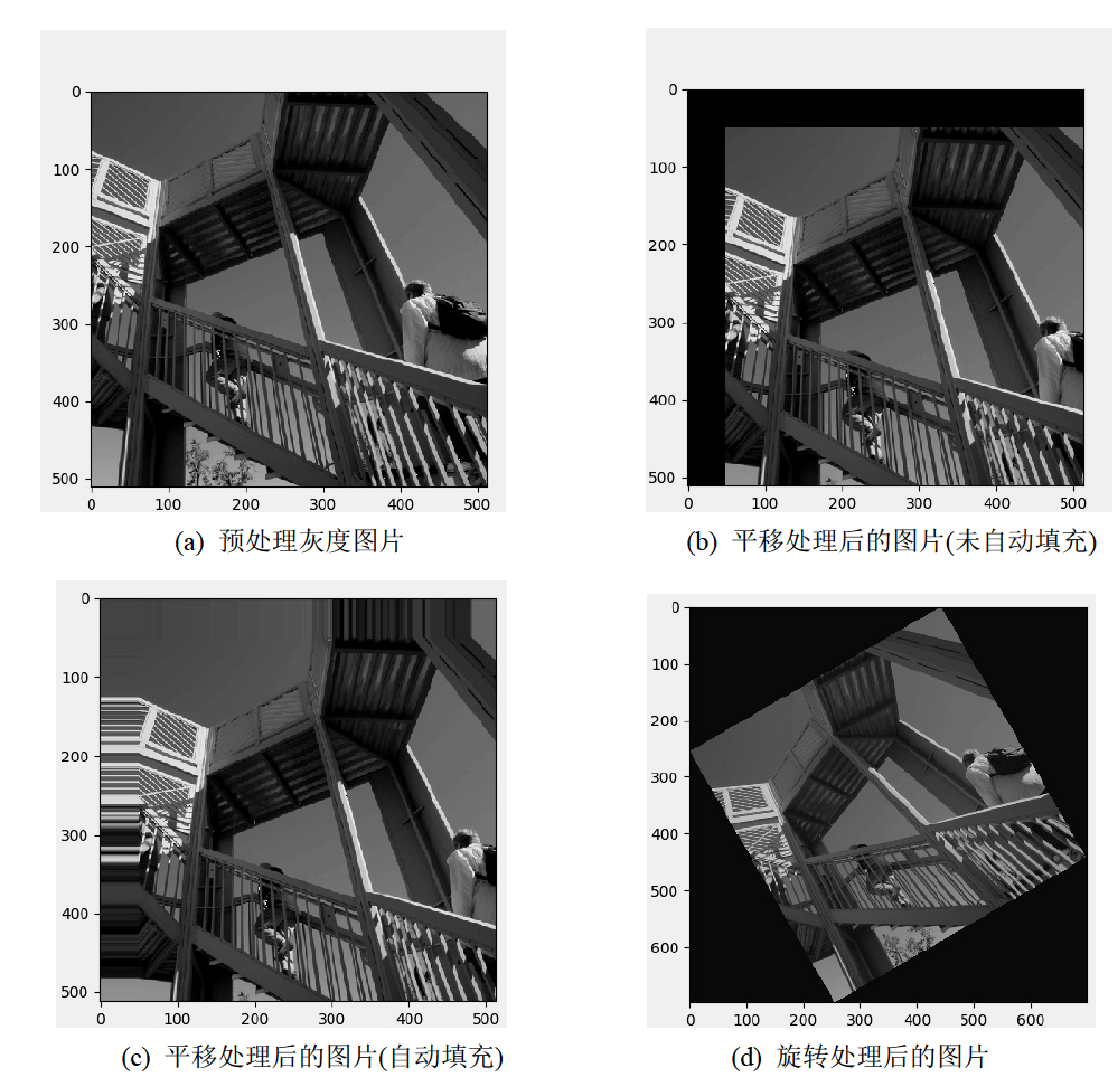
shifted ascent = ndimage.shift(ascent, (50, 50))
shifted ascent2 = ndimage.shift(ascent, (50, 50), mode="nearest")
rotated ascent = ndimage.rotate(ascent, 30)

pl.imshow(ascent, cmap=pl.cm.gray)
pl.figure()
pl.imshow(shifted ascent, cmap=pl.cm.gray)
pl.figure()
pl.imshow(shifted ascent2, cmap=pl.cm.gray)
pl.figure()
pl.imshow(rotated ascent, cmap=pl.cm.gray)
pl.show()
```

其中，ascent = misc.ascent()生成了一个 SciPy 库自带的灰度图片；语句 ndimage.shift (ascent, (50, 50))对图片进行了平移处理，处理后的图片如图 9-1(b)所示；语句 ndimage.shift (ascent, (50, 50), mode="nearest")为平移后自动填充，处理效果如图 9-1(c)所示；语句 rotated_ascent = ndimage.rotate(ascent, 30)对图片做了逆时针旋转 30° 的处理，处理后的图片如图 9-1(d)所示。

我们知道，无论是方程组求解，还是图像处理，使用一般的高级语言按部就班地编程，都需要耗费很长的时间，编出的程序也很冗长，还不得不考虑各种例外情况。Python 编程与之不同，它的理念是尽可能地运用各种第三方扩展库，迅速开发出符合要求的程

序。从上面的两个例题的编程可以看出，如何求方程组的解，如何对图像进行平移、填充、旋转等处理，我们不必关心，我们只关心业务逻辑，即关心我们要解决的问题。



(a) 预处理灰度图片　　(b) 平移处理后的图片(未自动填充)

(c) 平移处理后的图片(自动填充)　　(d) 旋转处理后的图片

图 9-1　使用 SciPy 库对图片进行处理

换言之，用 Python 编程，我们只管 What to do，不管 How to do，Python 编程的魅力由此可见一斑。

9.3.3　Matplotlib 库

前两节主要介绍了用于科学计算的 NumPy 库和 SciPy 库。但是，这两种库均未提供数据可视化工具。所以，这里我们引入 Matplotlib 库来解决可视化问题。

Matplotlib 是 Python 的一个 2D 绘图库，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形。通过使用 Matplotlib，开发者仅需要写几行代码便可以绘图，生成直方图、功率谱、条形图、错误图、散点图等，也可以绘制一些简单的 3D 图。

Matplotlib 的安装与上面的两个模块相似，同样可以使用 pip 安装或者自行安装，在 Ubuntu 环境下也可以通过如下命令进行安装：

```
sudo apt-get install Python-matplotlib
```

安装成功后，可以用如下命令测试是否安装成功：

```
import matplotlib
print(matplotlib.__version__)
```

如果安装成功，则会显示 Matplotlib 的版本号。

安装成功 Matplotlib 库后，就可以借助于它将运算结果“画”出来了。下面的例子将介绍 Matplotlib 的一些基本用法。

【例 9-7】使用 Matplotlib 进行画图的一些基本代码：

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 1000)  #设置自变量格式
y = np.sin(x) + 1            #设置因变量 y
z = np.cos(x**2) + 1          #设置因变量 z

plt.figure(figsize=(8, 4))    #设置图像大小
plt.plot(x, y, label="sinx+1", color='red', linewidth=2)
  #作图(x,y)，设置标签格式

plt.plot(x, z, label="cosx^2+1")  #作图(x, z)
plt.xlabel('Time(s)')  #设置 x 轴名称
plt.ylabel('Volt')  #设置 y 轴名称
plt.title('A simple Example')  #设置表格标题
plt.ylim(0, 2.2)  #显示的 y 轴范围
plt.legend() #显示图例
plt.show()  #显示作图结果
```

运行上面的代码，会弹出一个如图 9-2 所示的窗口。

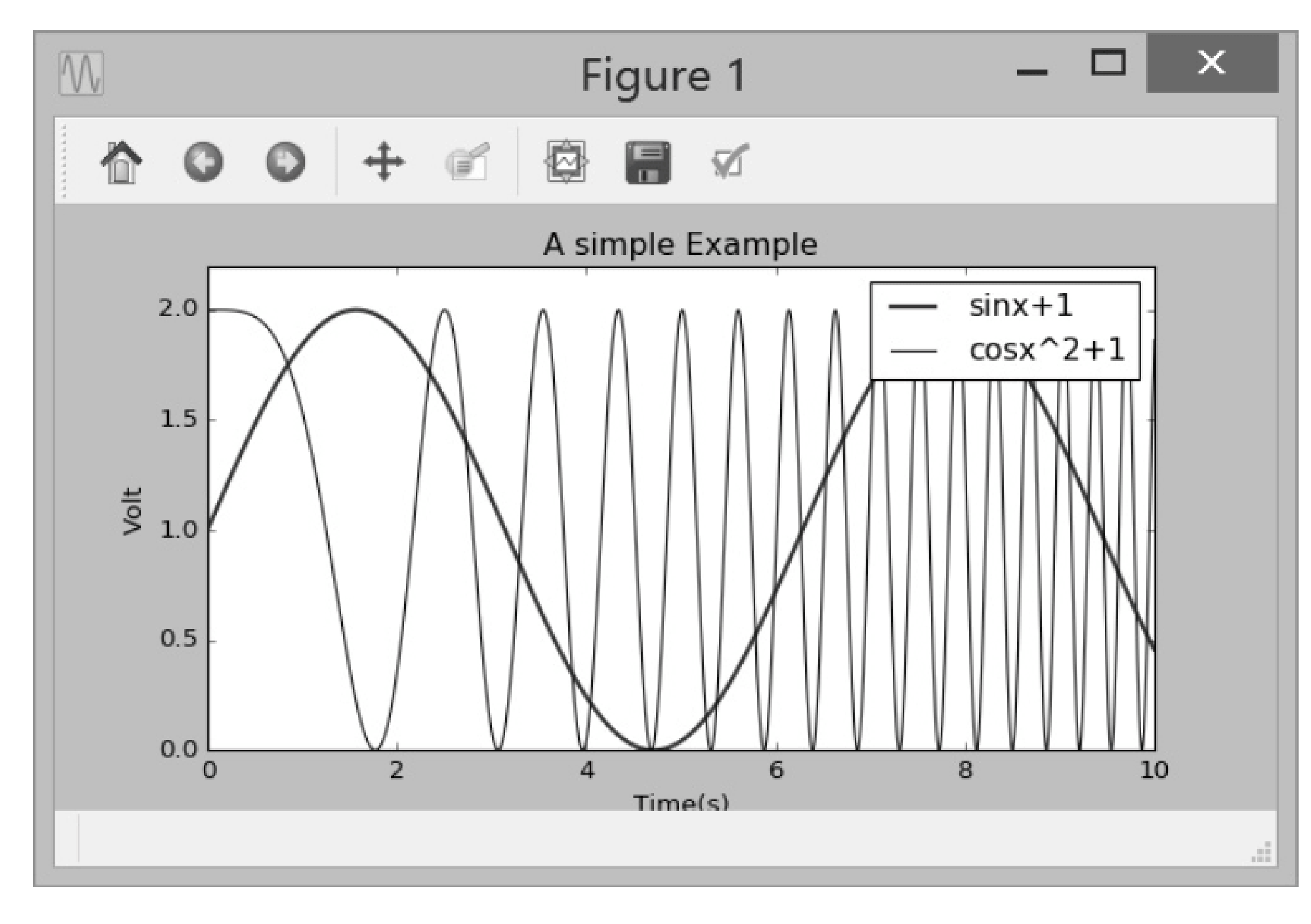


图 9-2　使用 Matplotlib 画图

单击保存图标，可以将表格保存为.eps、.jpg、.pdf、.png、.raw 等图片格式。保存的图片质量几乎可以满足各种出版要求。

注意：

- 实际绘图时，可能会发现有中文无法正常显示的情况，这是因为 Matplotlib 的默认字体为英文字体所致，需要在作图之前，指定默认字体为中文。代码如下：

  plt.rcParams[‘font.sans-serif’] = [‘SimHei’]
- 如果在保存作图图像时发现负号显示不正常，可以通过下面的代码解决：

  plt.rcParams[‘axes.unicode_minus’] = False

【例 9-8】使用 matplotlib 实现数据可视化：

```
import matplotlib.pyplot as plt
import xlrd

data= xlrd.open workbook("data.xls")
sh = data.sheet by name("Sheet1")
x=sh.col values(0)
y=sh.col values(1)
plt.plot(x, y, '.')
plt.show()
```

上面的代码是一个从 Excel 文件中读取数据，并使用 Python 实现数据可视化，其运行结果如图 9-3 所示。

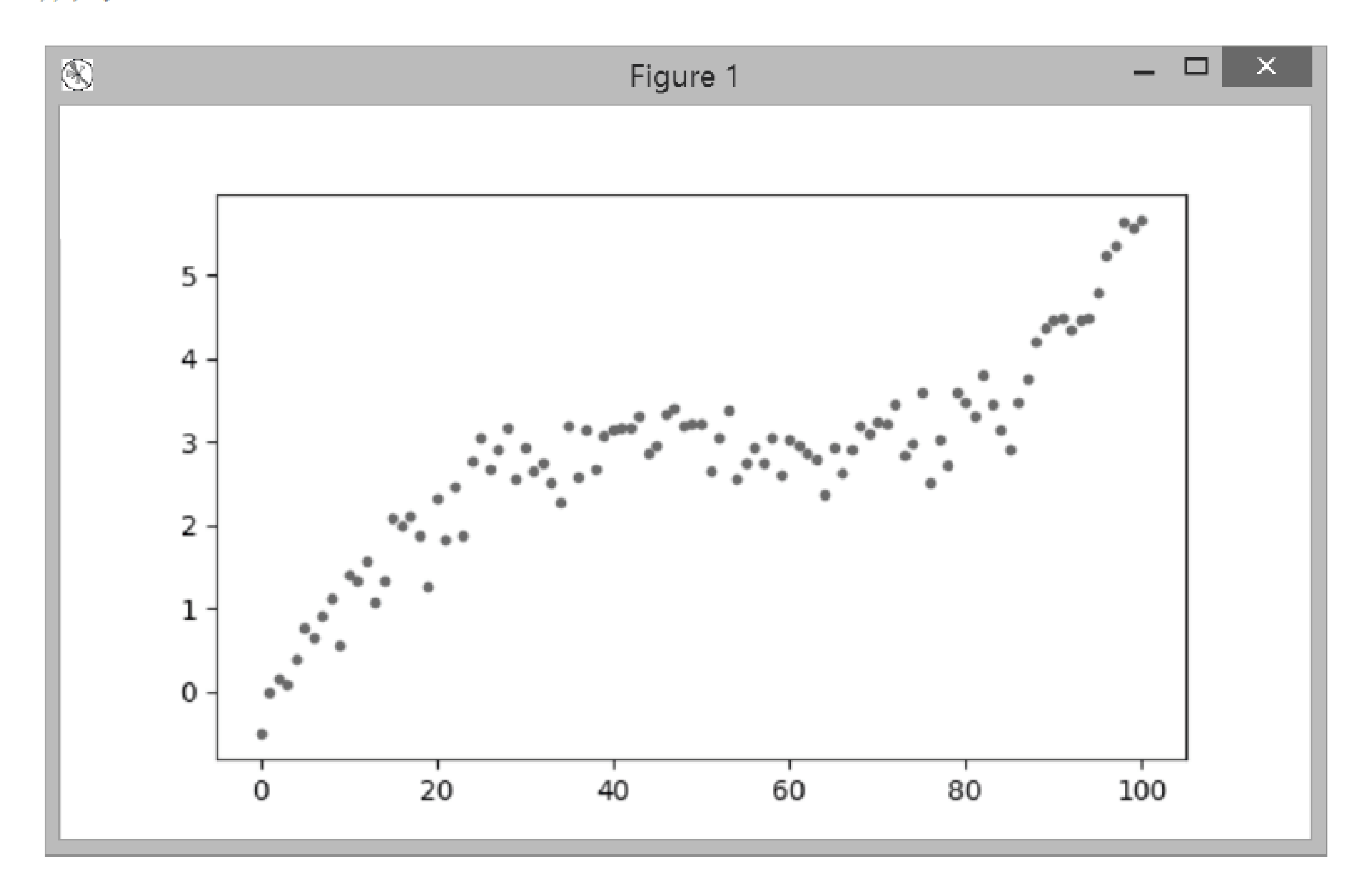


图 9-3　使用 Python 进行数据可视化

可以看到的是，使用 Python 语言，我们仅通过短短几行代码，即可实现数据的可视化显示，这是其他编程语言所不具有的优势。

9.3.4　Pandas 库

Pandas 库是 Python 下功能最为强大的数据分析和探索工具，也是本章介绍的这些库中

最为重要的，它包含高级的数据结构和精巧的分析工具，使得在 Python 中处理数据变得快速、简单。

Pandas 构建在 NumPy 之上，最初是作为金融数据分析用途而开发出来的，它是使 Python 成为强大而高效的数据分析环境的重要因素之一。

Pandas 的功能非常强大，支持类似 SQL 的数据增、删、改、查操作，并且包含了非常多的数据处理函数。

Pandas 为时间序列分析提供了很好的支持，可以很好地处理数据缺失等问题，可以灵活地对齐数据，解决不同数据源的数据集成时常见的问题。

1. 安装 Pandas

Pandas 的安装与上面的各个模块相似，同样可以使用 pip 安装或者自行安装，在 Ubuntu 环境下也可以通过如下命令进行安装：

```
sudo apt-get install Python-pandas
```

安装成功后，可以用如下命令测试是否安装成功：

```
import pandas as pd
print(pd.__version__)
```

如果安装正确，则会显示出 Pandas 的版本号。

2. Pandas 的数据结构 Series

Series 是一种类似于一维数组的对象，它由一组数据(各种 NumPy 数据类型)以及一组与之相关的数据标签(即索引)组成。

Series 的字符串表现形式为：索引在左边，值在右边。下面的例子展示了创建 Series 对象的几种方法。

【例 9-9】创建 Series 对象：

```
from pandas import Series    #从 Pandas 库中引用 Series

obj_list = [1, 2, 3, 4, 5]
obj_tuple = (1.2, 2.5, 3.3, 4.8, 5.4)
obj_dict = {'Tom': [16, 'boy'], 'Max': [12, 'boy'], 'Julia': [18,
'girl']}
series_list = Series(obj_list)
series_tuple =Series(obj_tuple, index=['a', 'b', 'c', 'd', 'e'])
series_dict = Series(obj_dict)
print("(1) 通过 list 建立 Series: ")
print(series_list)
print("(2) 通过 tuple 建立 Series: ")
print(series_tuple)
print("(3) 通过 dict 建立 Series: ")
print(series_dict)
```

输出结果：

```
(1) 通过list建立Series:
0    1
1    2
2    3
3    4
4    5
dtype: int64
(2) 通过tuple建立Series:
a    1.2
b    2.5
c    3.3
d    4.8
e    5.4
dtype: float64
(3) 通过dict建立Series:
Julia     [18, girl]
Max       [12, boy]
Tom       [16, boy]
dtype: object
>>>
```

可以看到，当没有显式地给出索引值的时候，Series 从 0 开始自动创建索引。相对于其他的很多数据结构来说，Series 结构最重要的一个功能是它可以在算术运算中自动对齐不同索引的数据。

【例 9-10】Series 的自动对齐功能：

```
from pandas import Series    #从 Pandas 库中引用 Series

obj_dict = {"王明": 7700, "张伟": 8600, "赵红": 9100, "郭强": 6700}
series_obj_1 = Series(obj_dict)
series_obj_2 = Series(obj_dict, index=['张伟', '郭强', '王明', '李洋'])
print(series_obj_1)
print("----------------")
print(series_obj_2)
print("----------------")
print(series_obj_1+series_obj_2)
```

输出结果：

```
张伟     8600
王明     7700
赵红     9100
郭强     6700
dtype: int64
----------------
张伟     8600.0
郭强     6700.0
王明     7700.0
李洋        NaN
dtype: float64
----------------
张伟     17200.0
李洋         NaN
王明     15400.0
赵红         NaN
郭强     13400.0
dtype: float64
>>>
```

从上面的例子中可以看到，在建立 series_obj_2 时，obj_dict 中跟索引值相匹配的值会被找出来赋给相应的索引，而“李洋”没在 obj_dict 中，所以对应的值被赋为 NaN(Not a Number)，即缺失值。在执行 series_obj_1 + series_obj_2 时，Series 对象自动对齐不同的索引值再进行计算。

3. Pandas 的数据结构 DataFrame

DataFrame 是 Pandas 的主要数据结构之一，是一种带有标签的二维对象，与 Excel 表格或者关系型数据库的结构十分相似。DataFrame 结构的数据都会有一个行索引和列索引，且每一列的数据格式可能是不同的。相对于 Series 来说，DataFrame 相当于多个带有相同索引的 Series 的组合，且每个 Series 都有一个不同的表头来识别不同的 Series。

【例 9-11】创建 DataFrame：

```
from pandas import DataFrame    #从 pandas 库中引用 DataFrame
from pandas import Series       #从 pandas 库中引用 Series

obj = {'name': ['Tom', 'Peter', 'Lucy', 'Max', 'Anne'], 'age': ['17',
'23', '44', '27', '36'],
      'status': ['student', 'student', 'doctor', 'clerk', 'performer']}
series_dict1 = Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
series_dict2 = Series([6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e'])
df_obj = DataFrame(obj) #创建 DataFrame 对象
df_obj2 = DataFrame([series_dict1,series_dict2])
print(df_obj)
print("----------------------")
print(df_obj2)
```

输出结果：

```
  age   name     status
0  17    Tom    student
1  23  Peter    student
2  44   Lucy     doctor
3  27    Max      clerk
4  36   Anne  performer
----------------------
   a  b  c  d   e
0  1  2  3  4   5
1  6  7  8  9  10
>>>
```

本例程序通过传入一个 NumPy 数组组成的字典来创建 DataFrame 对象，这是最为常用的方法。也可以利用多个具有相同索引的 Series 对象来创建 DataFrame 对象，不过，创建出的列表只能为横向列表。使用 df_obj.T 转置方法可将其转换成常用的纵向列表。

通过使用类似于访问类成员的方式，可以获取 DataFrame 对象指定的列数据(Series)或者新增列。

【例 9-12】DataFrame 的基本操作：

```
from pandas import DataFrame    #从 Pandas 库中引用 DataFrame
from pandas import Series       #从 Pandas 库中引用 Series

obj = {'name': ['Tom', 'Peter', 'Lucy', 'Max', 'Anne'], 'age': ['17',
'23', '44', '27', '36'],
      'status': ['student', 'student', 'doctor', 'clerk', 'performer']}
series_dict1 = Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
series_dict2 = Series([6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e'])
df_obj = DataFrame(obj) #创建 DataFrame 对象
```

```
df_obj2 = DataFrame([series_dict1,series_dict2])

print("---查看前几行数据，默认 5 行---")
print(df_obj.head())
print("---------提取一列-----------")
print(df_obj.age)
print("----------添加列------------")
df_obj['gender'] = ['m', 'm', 'f', 'm', 'f']
print(df_obj)
print("----------删除列------------")
del df_obj['status']
print(df_obj)
print("-----------转置-------------")
print(df_obj2.T)
```

输出结果：

```
---查看前几行数据，默认5行---
   age   name     status
0  17     Tom    student
1  23  Peter    student
2  44   Lucy     doctor
3  27    Max      clerk
4  36   Anne  performer
---------提取一列-----------
0     17
1     23
2     44
3     27
4     36
Name: age, dtype: object
----------添加列------------
   age   name     status gender
0  17     Tom    student      m
1  23  Peter    student      m
2  44   Lucy     doctor      f
3  27    Max      clerk      m
4  36   Anne  performer      f
----------删除列------------
   age   name gender
0  17     Tom      m
1  23  Peter      m
2  44   Lucy      f
3  27    Max      m
4  36   Anne      f
-----------转置-------------
   0   1
a  1   6
b  2   7
c  3   8
d  4   9
e  5  10
>>>
```

还有一些其他的常用 DataFrame 操作，如表 9-3 所示。

本书第 5 章介绍了用 Python 读取 CSV、Excel 文件的方法，但需要特别指出的是，Pandas 也可以从 Excel、CSV 等文件中读取或写入数据。不过默认的 Pandas 库并不能直接对 Excel 文件进行操作，还需要安装 xlrd(读入操作)和 xlwt(写入操作)库才能支持对 Excel 文件的读写。安装 xlrd 和 xlwt 库的方法如下：

```
pip install xlrd
pip install xlwt
```

安装完成后，就可以在 Pandas 中进行 Excel、CSV 等文件的操作了。

```
df obj2 = DataFrame([series dict1,series dict2])

print("---查看前几行数据，默认 5 行---")
print(df obj.head())
print("---------提取一列-----------")
print(df obj.age)
print("----------添加列------------")
df obj['gender'] = ['m', 'm', 'f', 'm', 'f']
print(df obj)
print("----------删除列------------")
del df obj['status']
print(df obj)
print("-----------转置-------------")
print(df_obj2.T)
```

输出结果：

```
---查看前几行数据，默认5行---
   age   name     status
0  17     Tom    student
1  23   Peter    student
2  44    Lucy     doctor
3  27     Max      clerk
4  36    Anne  performer
---------提取一列-----------
0    17
1    23
2    44
3    27
4    36
Name: age, dtype: object
----------添加列------------
   age   name     status gender
0  17     Tom    student      m
1  23   Peter    student      m
2  44    Lucy     doctor      f
3  27     Max      clerk      m
4  36    Anne  performer      f
----------删除列------------
   age   name gender
0  17     Tom      m
1  23   Peter      m
2  44    Lucy      f
3  27     Max      m
4  36    Anne      f
-----------转置-------------
   0   1
a  1   6
b  2   7
c  3   8
d  4   9
e  5  10
>>> 
```

还有一些其他的常用 DataFrame 操作，如表 9-3 所示。

本书第 5 章介绍了用 Python 读取 CSV、Excel 文件的方法，但需要特别指出的是，Pandas 也可以从 Excel、CSV 等文件中读取或写入数据。不过默认的 Pandas 库并不能直接对 Excel 文件进行操作，还需要安装 xlrd(读入操作)和 xlwt(写入操作)库才能支持对 Excel 文件的读写。安装 xlrd 和 xlwt 库的方法如下：

```
pip install xlrd
pip install xlwt
```

安装完成后，就可以在 Pandas 中进行 Excel、CSV 等文件的操作了。

表 9-3　其他的常用 DataFrame 操作

操　作	说　明
df_obj.dtypes	查看各行的数据格式
df_obj.tail()	查看后几行的数据，默认后 5 行
df_obj.index	查看索引
df_obj.columns	查看列名
df_obj.values	查看数据值
df_obj.describe	描述性统计
df_obj.sort(columns = ‘’)	按列名进行排序
df_obj.sort_values	多列排序
f_obj[‘列索引’]	显示列名下的数据
df_obj[1:3] #	获取 1~3 行的数据(切片操作)
df_obj.reindex()	根据 index 参数重新进行排序

【例 9-13】读入 Excel 文件：

```
import pandas as pd

df_obj = pd.read_excel('sult.xls')
print(type(df_obj))
print(df_obj.head())
```

输出结果：

```
<class 'pandas.core.frame.DataFrame'>
   P+(mV)  lgC+(mA)     10mV区间斜率    50mV区间平均斜率
0  -120.0  0.352260  1222.163105  1192.456669
1  -110.0  0.360442  1401.030899  1111.419158
2  -100.0  0.367580   761.145713   983.230411
3   -90.0  0.380718  1394.644111   965.479118
4   -80.0  0.387888  1183.299518   808.467750
>>> 
```

9.4　案例实训

本节通过两个案例介绍利用 Python 进行数据分析的一些基本方法，配以实际的数据进行讲解。

9.4.1　利用 Python 分析数据的基本情况——缺失值分析与数据离散度分析

实际的数据挖掘过程中，通过分析从客户手中得到的数据会发现，由于各种各样的原因，得到的数据有一部分是缺失的。如果缺失的数据量不大，手工解决这一问题并非难事，但实际情况往往相反，庞大的数据量和较多的属性值，依靠人工分辨的方法是非常不切实际的。所以，拟通过使用 Python 编写程序来帮助检测数据的缺失值、缺失个数等数据属性。利用 Pandas 库中的 describe()函数，可以将数据读入程序，轻松地完成上述操作。例如，现有如图 9-4 所示的日期/销量 Excel 表格。

	A	B
1	日期	销量
2	2015/3/1	51
3	2015/2/28	2618.2
4	2015/2/27	2608.4
5	2015/2/26	2651.9
6	2015/2/25	3442.1
7	2015/2/24	3393.1
8	2015/2/23	3136.6
9	2015/2/22	3744.1
10	2015/2/21	6607.4
11	2015/2/20	4060.3
12	2015/2/19	3614.7
13	2015/2/18	3295.5
14	2015/2/16	2332.1
15	2015/2/15	2699.3
16	2015/2/14	
17	2015/2/13	3036.8

图 9-4　日期/销售额 Excel 表格

若使用 Python 对数据的基本情况进行分析，可使用下面的程序：

```
import pandas as pd

print("例：使用 Python 分析数据基本信息。")
print("------------------------------")
while True:
    print("请输入数据文件所在路径 :")
    sale data = input()  #获取数据路径
    try:
        data = pd.read excel(sale data, index col='日期')
          #读取数据，指定“日期”列为索引

        print(data.describe())
        break
    except:
        print("文件打开失败，请确认路径")
```

运行上面的代码，输出结果如下：

```
例：使用Python分析数据基本信息。
------------------------------
请输入数据文件所在路径 :
catering_sale.xls
                销量
count   200.000000
mean   2755.214700
std     751.029772
min      22.000000
25%    2451.975000
50%    2655.850000
75%    3026.125000
max    9106.440000
>>> |
```

其中，count 表示的是非空数据总数，通过对比 len(data)的值可以得出缺失值的个数。其他的几个值分别为数据平均值(mean)、标准差(std)、最小值(min)、下四分位数(25%)、中位数(50%)、上四分位数(75%)和最大值(max)。在这 7 个数中，前两个数反映了数据的中心趋势，后 5 个数反映了数据的离散度，在数理统计中称为“五数概括”。

9.4.2　使用箱形图检测异常值——离群点挖掘

异常值指的是样本数据中个别明显偏离实际的点，亦称离散点或孤立点，因此异常值分析也叫作离群点分析或孤立点分析。

对数据进行异常值分析是为了检测数据中是否有录入错误或者不合乎常理的数据。这些异常的数值在后续数据挖掘中往往是十分危险的，极有可能造成数据扭曲。

这意味着，如果不加剔除地使用这些异常值，最后数据挖掘得到的结果可能和实际大相径庭。同时，挑出这些异常的数值并对其进行分析，也往往可以成为数据挖掘的突破口(如欺诈甄别等)。

箱形图(box-plot)又称为盒型图、盒式图、盒图或箱线图，是用来显示一组数据离散情况的统计图。

箱形图提供了一个用于识别异常值的标准：异常值被定义为小于 Q1−1.5IQR 或大于 Q3+1.5IQR 的值，其中，Q1 表示下四分位数，全部数据有四分之一小于这个值；四分位距(Inter Quartile Range，IQR)表示四分位数间距，是上四分位数和下四分位数之差，包含了全部数据的一半；Q3 表示上四分位数，全部数据有 1/4 大于这个值。

绘制箱形图依靠的是实际数据，不需要事先假定数据服从某种特定的分布形式，也不必对数据做任何限制性要求，它只是真实、直观地表现数据形状的本来面貌；另一方面，箱形图判断异常值的标准以四分位数和四分位距为基础，四分位数具有一定的耐抗性，多达 25%的数据可以变得任意远而不会很大地扰动四分位数，所以异常值不会对这个标准施加影响。箱形图识别异常值的结果比较客观，因此，它在识别异常值方面有一定的优越性。图 9-5 是箱形图的示意图。

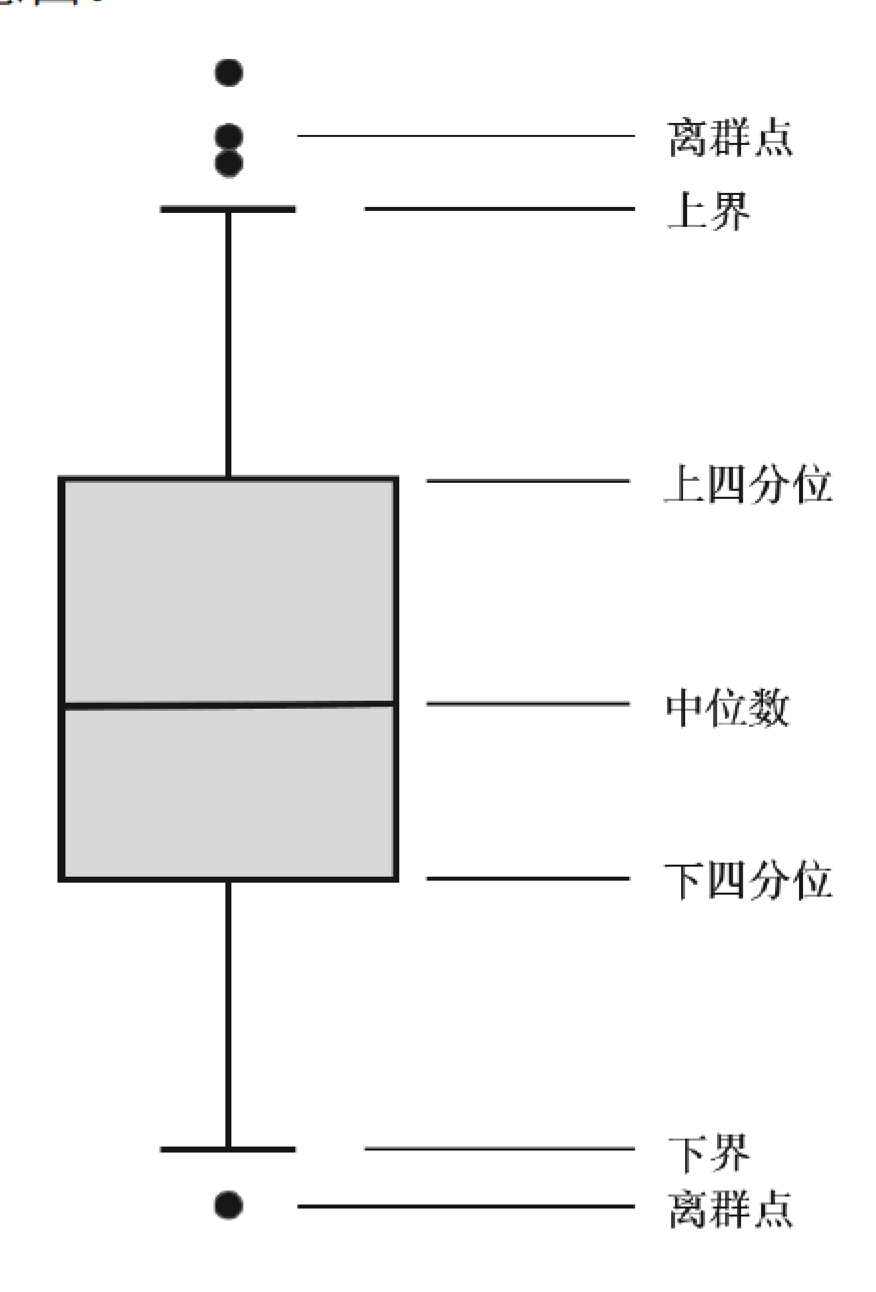


图 9-5　箱形图示意图

假定用于分析的数据包含属性 age，其值如下(按递增序)：13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70。使用箱形图检测离群点的程序如下：

```
import pandas as pd
from pandas import DataFrame
import matplotlib.pyplot as plt #导入图像库

value = [13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33,
33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70]
data = DataFrame(value)
plt.figure() #建立图像
p = data.boxplot(return type='dict') #画箱形图，直接使用 DataFrame 的方法
x = p['fliers'][0].get xdata() #'flies'即为异常值的标签
y = p['fliers'][0].get ydata()
y.sort() #从小到大排序，该方法直接改变原对象
#用 annotate 添加注释
#其中有些相近的点，注解会出现重叠，难以看清，需要一些技巧来控制。
#以下参数都是经过调试的，需要具体问题具体调试
for i in range(len(x)):
    if i>0:
        plt.annotate(y[i], xy = (x[i],y[i]), xytext=(x[i]+0.05 -0.8/(y[i]-
y[i-1]),y[i]))
    else:
        plt.annotate(y[i], xy = (x[i],y[i]), xytext=(x[i]+0.08,y[i]))
plt.show() #展示箱形图
```

运行上面的代码，会显示出如图 9-6 所示的箱形图。在图中，一个离群点 70 被明显地标出。在实际问题中，实验数据量特别巨大时，使用箱形图可以帮助我们快速地找出所有的离群点。

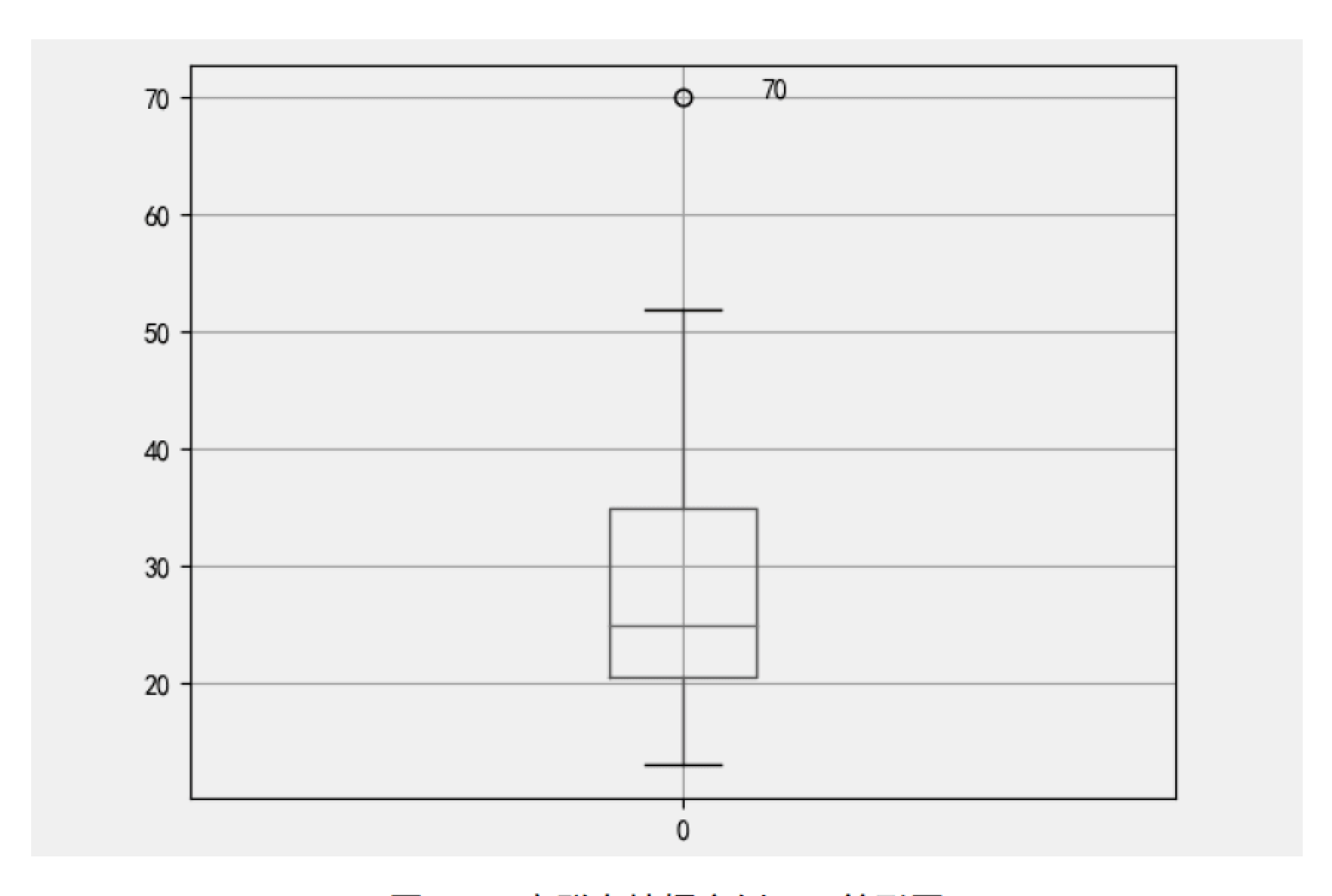


图 9-6　离群点挖掘实例——箱形图

本章 9.1 节曾提及，数据挖掘的最后一个步骤是知识表示，一般使用可视化的方式表示，以适合不同背景用户的需要。

针对本例而言，使用箱形图表示离群点，其本身就是数据的可视化。不难看出，数理统计中反映数据离散特征的“五数概括”在这里以可视化的方式进行了非常形象的表示，展示了数据的分布趋势。

本 章 小 结

由于篇幅所限，本章只是简单地介绍了 Python 在数据挖掘方面的简单应用，向读者介绍了用于数据分析与数据挖掘的 NumPy 库、SciPy 库、Pandas 库、Matplotlib 库等，并简单地展示了如何使用 Python 实现数据的可视化。目前使用 Python 进行数据挖掘十分热门，读者如果对本节内容感兴趣，可以参考《利用 Python 进行数据分析》一书(参考文献[1])，里面对 NumPy 库和 Pandas 库有着详细的介绍。

习　　题

1. 填空题

(1) SciPy 库依赖于(　　　)库，因此在安装 SciPy 库之前需要先安装好它。

(2) 欲生成由 0，1，2，…，8 组成的 3 行 3 列数组，请完善下面的程序：

```
import numpy as np
data =
arr = ________________________________
```

(3) 请将下面的代码补全，并写出运行结果：

```

s1 = Series([7.3, -2.5, 3.4, 1.5],index = ["a","c","d","e"])
s2 = Series([-2.1, 3.6, -1.5, 4, 3.1],index = ["a","c","e","f","g"])
print(s1+s2)
```

运行结果为：

```
a   5.2
c   ___
d
e
f
g   ___
```

2. 选择题

(1) 下面代码的输出结果为(　　)：

```
import numpy as np
arr3 = np.eye(3)
```

A. [[1. 0. 0]
[0. 1. 0]
[0. 0. 1]]

B. [[0. 0. 0]
[0. 0. 0]
[0. 0. 0]]

C. [[1. 1. 1]
[1. 1. 1]
[1. 1. 1]]

D. [[1. 0. 0]
[0. 2. 0]
[0. 0. 3]]

(2) 对 DataFrame 对象进行操作时，下面(　　)语句可实现查看前面 5 行数据的功能？

A. DataFrame.align

B. DataFrame.age

C. DataFrame.head

D. DataFrame.shape

3. 实验操作题

(1) 使用 NumPy 库生成一个由 0~14 组成的 3 行 5 列的数组。

(2) 使用 NumPy 库计算数组[[1,1], [0,1]]乘以数组[[2,0], [3,4]]的结果并输出。

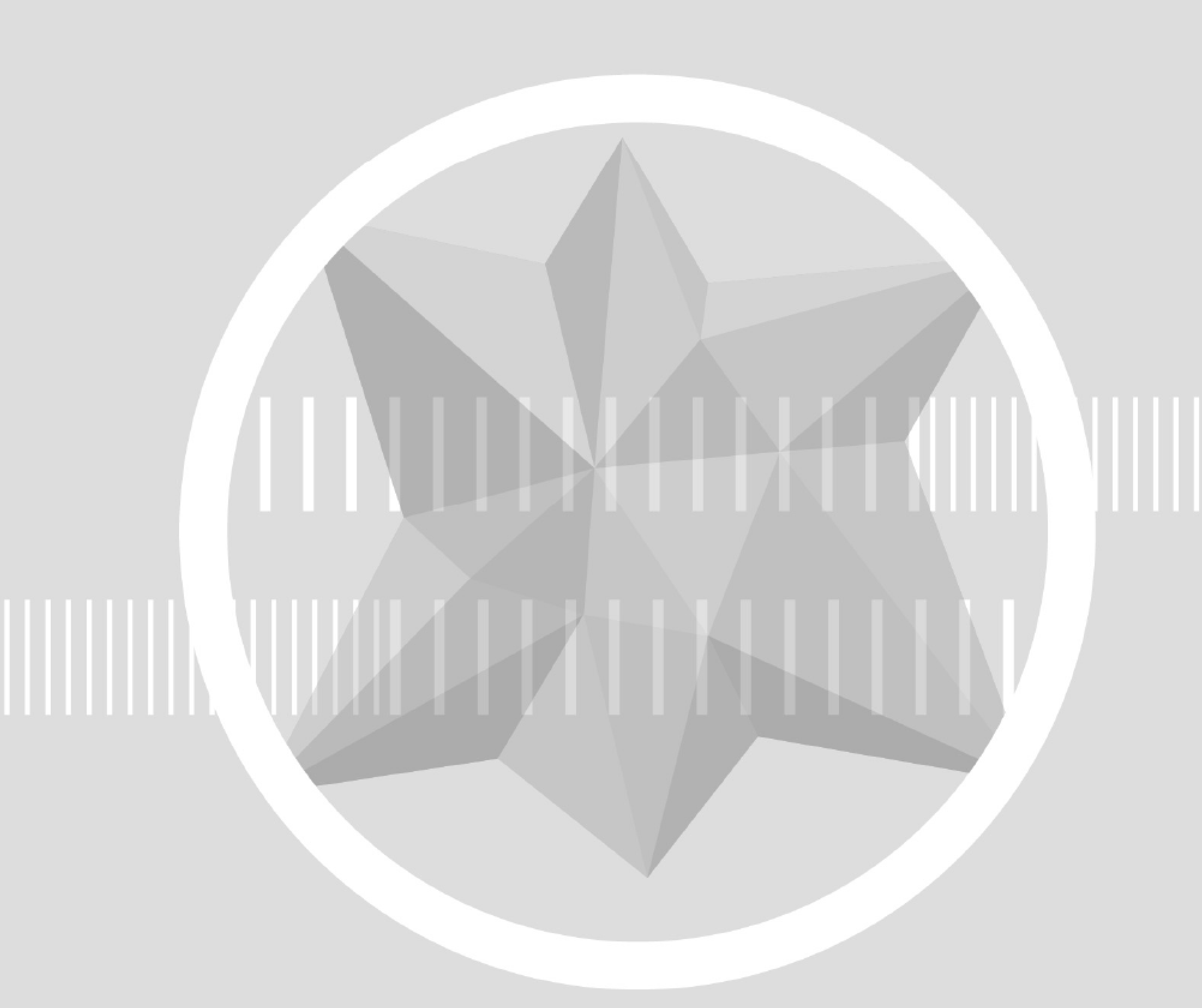
(3) 使用 Pandas 库对数组[[1,2,3,4,5],[6,7,8,9,10]]以 0、1 为行号，以字母为列号建立索引，并输出运行结果和转置后的结果。

(4) 现有矩阵：

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 1 \\ 3 & 4 & 3 \end{bmatrix}$$

利用 SciPy 库求矩阵的逆，并写出代码和输出结果。

(5) 从 0 开始，步长值为 1 或者-1，且出现的概率相等，通过使用 Python 内置的 random 实现 1000 步的随机漫步，并使用 Matplotlib 生成折线图。



第 10 章

GUI 编程和用户界面

本章要点

(1) GUI 界面的概念。
(2) Tkinker 模块。
(3) Tkinker 的各种组件。
(4) 网格布局管理器。
(5) GUI 编程。

学习目标

(1) 了解什么是 GUI 界面。
(2) 掌握 Tkinker 及其各个组件的应用方法。
(3) 掌握网格布局管理器的相关知识。
(4) 掌握 GUI 编程方法。

前面各章的程序都是基于文本用户界面(Text-based User Interface，TUI)的。当然，TUI 界面有着简洁、稳定、资源消耗较小等优点，但是，对于不懂编程的用户来说，TUI 界面显得不甚美观，操作也不甚方便。为了使输入输出更加直观，使操作方式更加简易，就需要使用图形用户界面(Graphical User Interface，GUI)，或称图形用户接口。

图形用户界面是人与计算机通信的一种界面显示格式，允许用户使用鼠标等输入设备操纵屏幕上的图标或菜单选项，以选择命令、调用文件、启动程序或执行其他一些任务。与通过键盘输入文本或字符命令来完成例行任务的文本界面相比，图形用户界面有许多优点。Python 提供了很多的 GUI 界面工具，如 Python 的标准 Tk GUI 工具包接口 Tkinter、wxWidgets 模块、easyGUI 模块、wxPython 模块等。本章主要介绍使用 Tkinter 模块开发图形用户界面的方法，并介绍一些常用的 Tkinker 组件。根据实际情况选择合适的模块来实现所需的功能，也是减少编程工作量必不可少的方法。

10.1　Tkinter 模块

Tkinter 模块(Tk 接口)是 Python 的标准 Tk GUI 工具包的接口。Tk 和 Tkinter 可以在大多数 Unix 平台下使用，当然也可以应用在 Windows 和 Macintosh 系统里。图 10-1 所示的是一个简单的房屋按揭利率计算器，这一程序有三个用户输入项和一个输出项。图 10-1(a)是 TUI 程序及其输出结果，图 10-1(b)是使用 Tkinter 控件的 GUI 输入输出界面。

在图 10-1(b)的界面中有三个白色的输入文本框，用户可以在输入文本框中单击鼠标，输入新的数据或者更改现有的数据。在输入完所有的数据后，单击界面下方的“计算每月应还款金额”按钮，就会将数据提交给程序并计算出相应的还款金额，并显示在下面的框体中。图中的文本框称为输入框控件(Entry Widget)，左边显示的文本信息称为标签控件(Label Widget)，下面的提交按钮称为按钮控件(Button Widget)。

除了这些控件，Tkinter 还提供了诸如列表框组件、画布组件、复选框组件、菜单组件等，本章后面将对其做详细介绍。

为了术语的统一，以下我们将“组件”和“控件”统称为“组件”。

```
def main():
    # 房贷计算器
    Principal = float(input("请输入贷款金额："))
    year_rate = float(input("请输入贷款利率(%)："))
    month_rate = year_rate/12  # 转换成月利率
    years = float(input("请输入贷款期限(年)："))
    months = years * 12  # 转换成月数
    sum = (Principal * (month_rate/100) * pow((1 +
month_rate/100),months))\
          /(pow((1 + month_rate/100),months)-1)  # 月
还款计算公式
    print("每月应还款金额：" + "%.2f"%sum)

main()
```

```
请输入贷款金额：300000
请输入贷款利率(%)：5.51
请输入贷款期限(年)：10
每月应还款金额：3257.28
```

(a) TUI 界面



(b) GUI 界面

图 10-1　TUI 界面与 GUI 界面的比较

10.1.1　创建 Windows 窗体

在 GUI 程序中，首先需要建立一个顶层窗口，这个顶层窗口可以容纳所有的小窗口对象，如标签、按钮、列表框等。也就是说，顶层窗口是用来放置其他窗口或组件的地方。

1. 创建窗口对象

用下面的语句可以创建一个顶层窗口，或者叫根窗口(有的文献称为主窗口)：

```
import tkinter
win = tkinter.Tk()
```

其中第一行代码用于导入 Tkinter 模块，第二行代码用于创建一个 Windows 窗体实例对象并命名为 win。在窗体对象创建完成后，可以使用以下的代码来显示窗体：

```
win.mainloop()
```

同时，mainloop()函数将进入无限监听事件循环，直到单击窗体右上方的关闭按钮，或者使用其他方法将窗口关闭。

【例 10-1】显示一个 Windows 窗体：

```
import tkinter
win = tkinter.Tk()
win.mainloop()
```

运行程序后，会显示图 10-2 所示的窗口。

2. 设置窗体属性

可以通过设置窗体的属性来改变窗体的显示方式。创建完窗体对象后，可以用 title()方法来设置窗口的标题，如下面的例子。

【例 10-2】设置窗口标题：

```
import tkinter
win = tkinter.Tk()
win.title("新建窗口")
win.mainloop()
```

运行代码后，会显示图 10-3 所示的有标题的窗口。

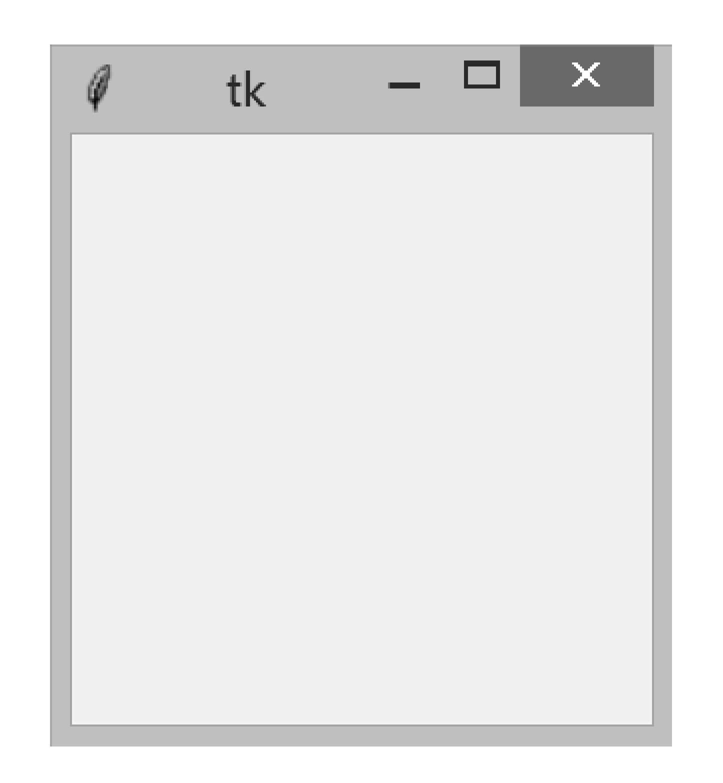


图 10-2　一个 Windows 窗口



图 10-3　有标题的窗口

还可以通过内建的 geometry()、maxsize()和 minsize()方法设置窗口的大小。

geometry(size)方法设置窗体大小，size 为指定的大小，其中，size 的格式为“宽度 x 高度”(注意，这里的“x”不是乘号，而是英文字母)；maxsize()和 minsize()方法用于设置最大窗体和最小窗体的尺寸，格式如下：

```
win.geometry("宽度 x 高度")
win.maxsize(宽度, 高度)
win.minsize(宽度, 高度)
```

【例 10-3】设置 Windows 窗口尺寸：

```
import tkinter
win = tkinter.Tk()
win.geometry("1024x768")
win.minsize(800,600)
win.maxsize(1440,900)
win.mainloop()
```

本例显示了一个 Windows 窗口，将其初始大小设置为 1024×768，并设置最小为 800×600，最大为 1440×900。

10.1.2　标签组件 Label

Label 组件是最简单的组件之一，用于在窗口中显示文本或位图。下面就是一个使用 Label 组件的简单例子。

【例 10-4】使用 Label 创建一个标签：

```
import tkinter
win = tkinter.Tk()
win.title("新建窗口")
Lab = tkinter.Label(win,text="label组件使用例子")
Lab.pack()
win.mainloop()
```

在上面的例子中，首先创建一个窗体，然后通过如下语句创建 Label 组件并设置显示的文本：

```
Label 对象 = Label(tkinter Windows 窗口对象, text=要显示的文本)
```

然后，通过.pack()方法显示 Label 组件。运行上面的代码，将会弹出一个如图 10-4 所示的窗口。

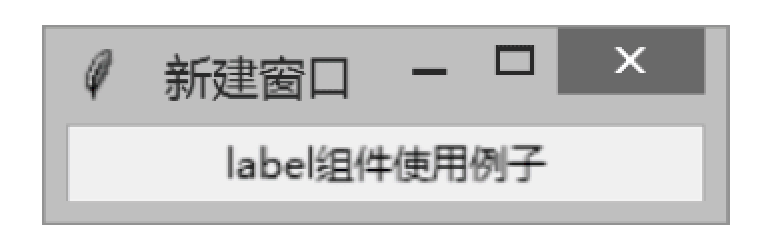


图 10-4　一个 label 组件

除了显示文本外，还可以运用 Label 组件的 bitmap 属性在窗口中显示自带的位图。代码如下：

```
L = Label(win, bitmap=图标)
```

其中“图标”的可选值如表 10-1 所示。

表 10-1　可选用位图

值	具体描述
Error	显示错误图标
Hourglass	显示沙漏图标
Info	显示信息图标
Questhead	显示疑问头像图标
Question	显示疑问图标
Warning	显示警告图标
gray12	显示灰度背景图标 gray12

续表

值	具体描述
gray25	显示灰度背景图标 gray25
gray50	显示灰度背景图标 gray50
gray75	显示灰度背景图标 gray75

【例 10-5】显示所有的可选位图：

```
import tkinter
win = tkinter.Tk()  #创建窗口对象
win.title("新建窗口")  #设置窗口标题

l1 = tkinter.Label(win,bitmap='error')  #显示错误图标
l1.pack()  #显示 Label 组件
l2 = tkinter.Label(win,bitmap='hourglass')  #显示沙漏图标
l2.pack()  #显示 Labe2 组件
l3 = tkinter.Label(win,bitmap='info')  #显示信息图标
l3.pack()  #显示 Labe3 组件
l4 = tkinter.Label(win,bitmap='questhead')  #显示疑问头像图标
l4.pack()  #显示 Labe4 组件
l5 = tkinter.Label(win,bitmap='question')  #显示疑问图标
l5.pack()  #显示 Labe5 组件
l6 = tkinter.Label(win,bitmap='warning')  #显示警告图标
l6.pack()  #显示 Labe6 组件
l7 = tkinter.Label(win,bitmap='gray12')  #显示灰度背景图标 gray12
l7.pack()  #显示 Labe7 组件
l8 = tkinter.Label(win,bitmap='gray25')  #显示灰度背景图标 gray25
l8.pack()  #显示 Labe8 组件
l9 = tkinter.Label(win,bitmap='gray50')  #显示灰度背景图标 gray50
l9.pack()  #显示 Labe9 组件
l10 = tkinter.Label(win,bitmap='gray75')  #显示灰度背景图标 gray75
l10.pack()  #显示 Label0 组件
win.mainloop()
```

运行后会显示如图 10-5 所示的窗口。

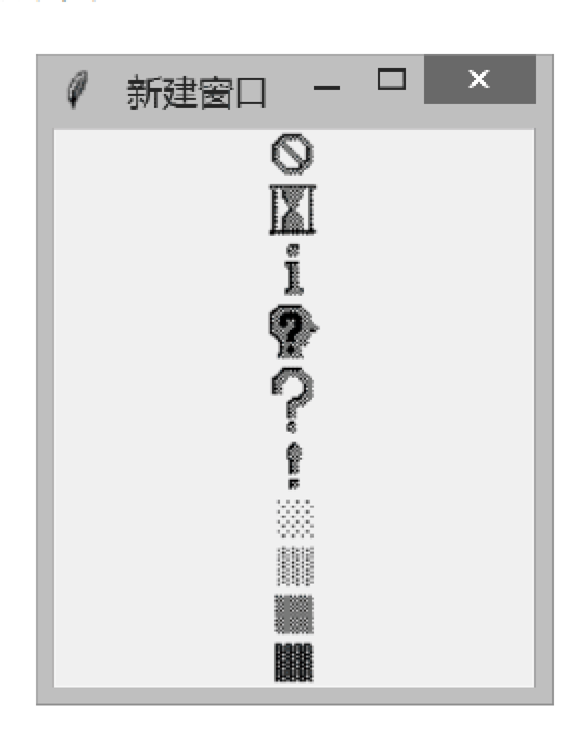


图 10-5　显示可选位图的窗口

由于内置的位图个数有限，而且显示的都是灰度图，所以，在实际的应用中，往往会



选择一些自定义的图标。这时，可以运用 image 属性和 bm 属性来设置自定义的图标，代码如下：

```
import tkinter
win = tkinter.Tk()  #创建窗口对象
win.title("新建窗口")  #设置窗口标题
bm = tkinter.PhotoImage(file = 图片名)
label = tkinter.Label(win,image = bm)
label.bm = bm
```

【例 10-6】使用 label 的 image 属性添加自定义图标：

```
import tkinter
win = tkinter.Tk()  #创建窗口对象
win.title("新建窗口")  #设置窗口标题
bm = tkinter.PhotoImage(file =
"D:\Python34\Lib\idlelib\Icons\idle 48.png")
label = tkinter.Label(win,image = bm)
label.bm = bm
label.pack()
win.mainloop()
```

运行上面的程序，可以看到，窗口中已经显示出自定义的图标，如图 10-6 所示。



图 10-6　添加自定义图标的窗口

除上面介绍的几个具体方法外，Label 组件还有一些常用的属性，如表 10-2 所示。

表 10-2　Label 组件的常用属性

属　性	说　明
fg	设置组件的前景色
bg	设置组件的背景色
width	设置组件宽度
height	设置组件高度
compound	设置文本或图像如何在 Label 上显示，默认值为 None。 当指定 image/bitmap 时，本文(text)将会被覆盖，只显示图像。可选值如下。 Left：图像居左显示。 Right：图像居右显示。 Top：图像居上显示。 Bottom：图像居下显示。 Center：图像居中显示

续表

属　性	说　明
wraplength	指定单行文本的长度，用于多行文本显示
justify	指定多行文本的对齐方式
anchor	指定文本或图片在 Label 中的显示位置。可选值如下。 e：垂直居中，水平居右。 w：垂直居中，水平居左。 n：垂直居上，水平居中。 s：垂直居下，水平居中。 也可是上面 4 个值的两两组合。 center：垂直居中，水平居中

10.1.3　按钮组件 Button

1. 创建和显示 Button 对象

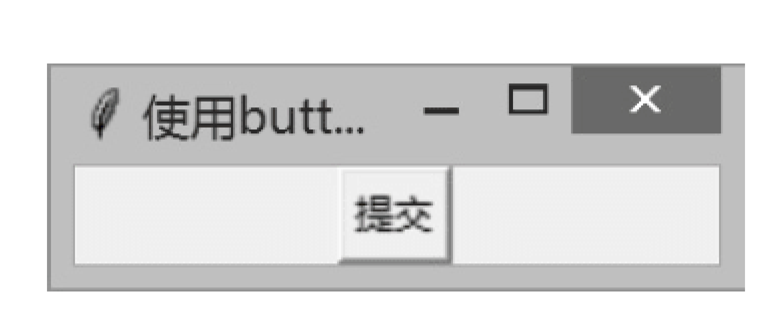
Button 组件用于在窗口中设置和显示按钮。创建 Button 对象的基本方法如下：

```
Button 对象 = Button(tkinter Windows 窗口对象, text = Button 显示名称,
command = 点击后调用)
Button 对象.pack()
```

【例 10-7】创建简单的按钮：

```
import tkinter
from tkinter import messagebox
def Submit():
    messagebox.showinfo(title = "",message = "提交")
win = tkinter.Tk()  #创建窗口对象
win.title("使用 button 组件的简单例子")  #设置窗口标题
b = tkinter.Button(win,text = "提交",command = Submit)  #创建 Button 组件
b.pack()  #显示 Button 组件
win.mainloop()
```

运行上面的程序，会显示如图 10-7(a)所示的窗口，在窗口中单击“提交”按钮，就会调用 Submit()函数，弹出如图 10-7(b)所示的提示窗口，单击“确定”按钮后将数据提交。



(a) 带有一个 Button 组件的窗口



(b) 提交确认窗口

图 10-7　一个 Button 组件

本例程序用到了消息框组件，10.1.4 小节将对此做详细的介绍。

2. Button 对象的常用属性

按钮上既可以显示文本，也可以显示用户自定义的图片。可以应用 image 属性和 bm 属性进行设置，其方法如下：

```
bm = PhotoImage(file = 图片名)
bt= Button(win, text = "图片", image = bm, command = 点击后调用)
bt.bm = bm
```

【例 10-8】创建图片格式的按钮：

```
import tkinter
from tkinter import messagebox

def Submit():
   messagebox.showinfo(title = "", message = "提交")

win = tkinter.Tk()  #创建窗口对象
win.title("使用 button 组件的简单例子")  #设置窗口标题
bm = tkinter.PhotoImage(file =
"D:\Anaconda3\Lib\idlelib\Icons\idle 48.png")
bt= tkinter.Button(win, text = "图片", image = bm, command = Submit)
bt.bm = bm
bt.pack()
win.mainloop()
```

运行上面的代码后，会显示如图 10-8 所示的窗口。

从图中可以看出，因为未设置按钮的大小和位置，所以按钮显示的位置不是十分合理。我们可以通过下面的代码对其进行设置。

【例 10-9】设置一个位置合理的按钮：

```
import tkinter
from tkinter import messagebox

def Submit():
   messagebox.showinfo(title = "",message = "提交")

win = tkinter.Tk()  #创建窗口对象
win.title("使用 button 组件的简单例子")  #设置窗口标题

bm = tkinter.PhotoImage(file =
"D:\Anaconda3\Lib\idlelib\Icons\idle 48.png")
label = tkinter.Label(win,image = bm)
label.bm = bm
label.pack()
b= tkinter.Button(win, text="确认",command=Submit, width=10, height=1,
compound="bottom")  #创建 button 对象
b.pack()
win.mainloop()
```

在上面的代码中，通过设置 Button 组件的 width 属性和 height 属性，改变了 Button 组件的大小，使其看起来更加美观。同时，通过设置 compound 属性，调整了 Button 组件的位置，效果如图 10-9 所示。

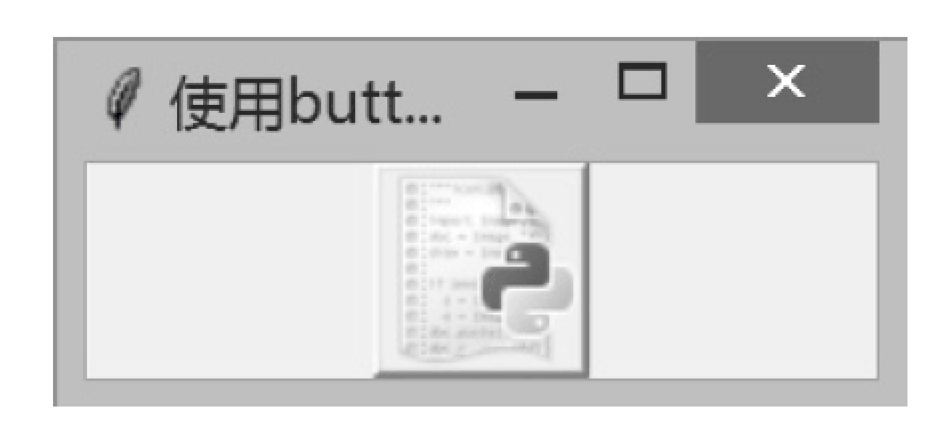



图 10-8　图片格式的 Button



图 10-9　位置合理的 Button

除了上述属性外，Button 组件还有一些其他的常用属性，如表 10-3 所示。

表 10-3　Button 组件的常用属性

属　性	说　明
fg	设置组件的前景色
bg	设置组件的背景色
compound	设置文本或图像如何在按钮上显示，默认值为 None。 当指定 Image/bitmap 时，本文(text)将会被覆盖，只显示图像。可用值如下。 Left：图像居左显示。 Right：图像居右显示。 Top：图像居上显示。 Bottom：图像居下显示。 Center：图像居中显示
wraplength	指定单行文本的长度，用于多行文本显示
bitmap	指定按钮显示位图
state	设置组件状态
bd	设置按钮边框大小，默认值为 1 或 2 个像素

【例 10-10】设置一个有边框的按钮：

```
import tkinter
from tkinter import messagebox

win = tkinter.Tk()  #创建窗口对象
win.title("使用 button 组件的简单例子")  #设置窗口标题
b1= tkinter.Button(win,text = "加粗按钮",bd = 10)  #创建 Button 对象
b2= tkinter.Button(win,text = "被禁用的按钮",state = "disabled")
  #创建 Button 对象

b1.pack()
b2.pack()
win.mainloop()
```

在例 10-10 中，我们设置了两个按钮，其中一个设置为加粗的边框，另一个设置为禁用状态，效果如图 10-10 所示。



图 10-10　有边框的 Button

10.1.4　消息框组件 Messagebox

弹出消息框是图形界面的一个基本功能，在图形界面的应用中也是最广的。在 tkinter 中，可以使用 tkinter.messagebox 模块来实现此功能。

1. 弹出一个提示消息框

使用 showinfo()函数可以弹出提示消息框，具体格式如下：

```
showinfo(title = 标题, message= 提示内容)
```

【例 10-11】使用 showinfo()弹出一个提示消息框：

```
from tkinter.messagebox import *
showinfo(title="提示", message="欢迎使用本系统")
```

运行上面的程序，会弹出一个如图 10-11 所示的提示消息框。单击“确定”按钮或者右上角的“×”按钮可关闭该提示消息框。



图 10-11　提示消息框

2. 弹出警告消息框

使用 showwarning()函数可以弹出警告消息框，方法如下：

```
showwarning(title = 标题, message = 内容)
```

【例 10-12】使用 showwarning()函数弹出一个警告消息框：

```
from tkinter.messagebox import *
showwarning(title="提示", message="请填写验证码")
```

运行程序，会弹出一个如图 10-12 所示的警告消息框。可以看出，警告消息框左侧的警告图标与上面的提示消息框图标不同。

3. 弹出错误消息框

使用 showerror()函数可以弹出错误消息框，方法如下：

```
showerror(title = 标题, message = 内容)
```

【例 10-13】使用 showerror()弹出一个错误消息框：

```
from tkinter.messagebox import *
showerror(title="提示", message="账号或密码错误")
```

运行上面的代码，会弹出一个如图 10-13 所示的错误消息框，左侧显示为错误图标。



图 10-12　警告消息框



图 10-13　错误消息框

4. 弹出疑问消息框

使用 askquestion()函数可以弹出一个包含“是(yes)”和“否(no)”按钮的疑问消息框，方法如下：

```
askquestion(title = 标题, message = 内容)
```

如果用户单击“是”按钮，则 askquestion()函数返回字符串“YES”；如果用户单击“否”按钮，则 askquestion()函数会返回字符串“NO”。

【例 10-14】使用 askquestion()函数弹出一个疑问消息框：

```
from tkinter.messagebox import *
ret = askquestion(title = "密码修改",message = "是否确认重置此密码？")
if ret == YES:
    showinfo(title="提示", message="密码已重置")
```

运行上面的代码，就会弹出一个如图 10-14 所示的疑问消息框。

也可以使用 askyesnocancel()实现上面的功能。与 askquestion()函数不同的是，用户单击“是”或“否”时，返回值为 True 或者 False。

5. 其他格式的疑问消息框

使用 askokcancel()函数，可以弹出一个包含“确定”和“取消”的疑问消息框，方法如下：

```
askokcancel(title = 标题, message = 内容)
```

用户单击“确定”或“取消”按钮时，askokcancel()函数会返回 True 或者 False。

【例 10-15】弹出一个带“确认”和“取消”按钮的疑问消息框：

```
from tkinter.messagebox import *
ret = askokcancel(title = "密码修改", message = "是否确认重置此密码？")
if ret == True:
    showinfo(title="提示", message="密码已重置")
```

运行上面的代码，显示如图 10-15 所示的疑问消息框，可见按钮已经变成了“确定”和“取消”。



图 10-14　带“是”和“否”的疑问消息框



图 10-15　带“确定”和“取消”的疑问消息框

使用 askretrycancel()函数，可以弹出一个包含“重试”和“取消”按钮的疑问消息框。方法如下：

```
askretrycancel(title = 标题, message = 内容)
```

【例 10-16】弹出一个带“重试”和“取消”按钮的警告消息框：

```
from tkinter.messagebox import *
ret = askretrycancel(title = "密码修改", message = "操作超时")
if ret == True:
    showinfo(title="提示", message="数据重置")
```

运行上面的代码，会弹出如图 10-16 所示的警告消息框，单击“重试”按钮，会弹出如图 10-17 所示的提示消息框。

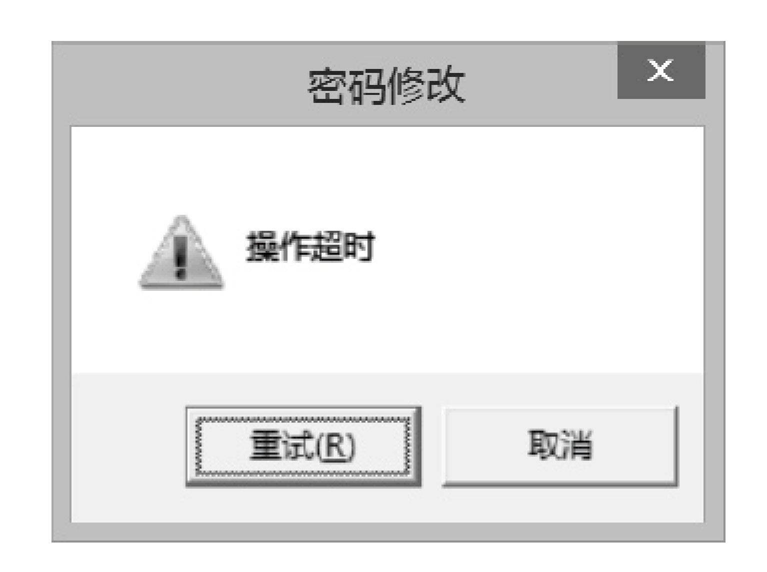


图 10-16　警告消息框



图 10-17　提示消息框

10.1.5　只读文本框 Entry

Entry 组件用于在窗口中输入单行文本。

1. 创建和显示 Entry 组件对象

创建 Entry 组件的方法如下：

```
Entry 对象 = Entry(tkinter  Windows 窗口对象)
Entry 对象.pack()
```

【例 10-17】使用 Entry 组件的简单例子：

```
import tkinter
win = tkinter.Tk()  #创建窗口对象
win.title("使用 Entry 组件的简单例子")  #设置窗口标题
entry = tkinter.Entry(win)  #创建 Entry 组件
entry.pack()
win.mainloop()
```

运行上面的代码，就会弹出一个如图 10-18 所示的窗口。

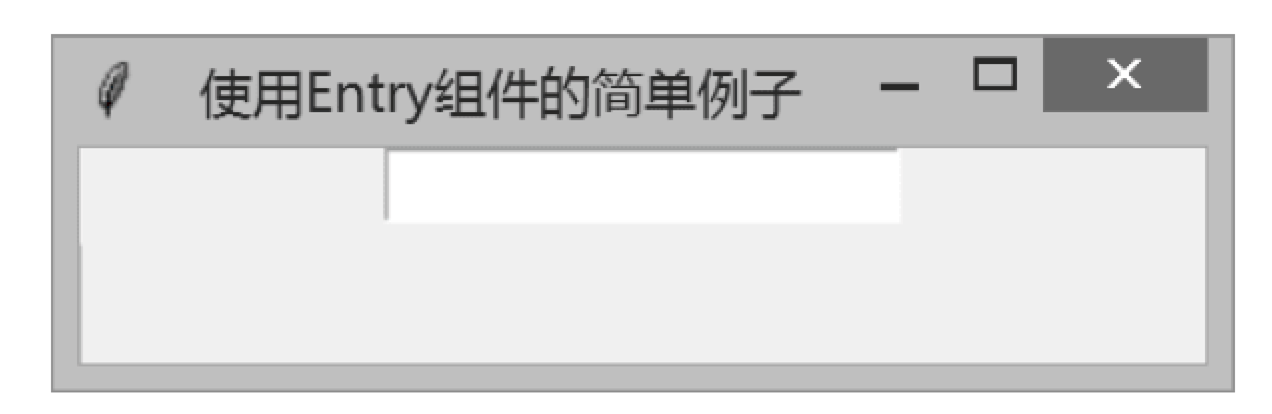


图 10-18　一个 Entry 组件

2. 获取 Entry 组件的内容

为了获取 Entry 组件的内容，需要使用 textvariable 属性为 Entry 组件指定一个对应的变量，例如：

```
import tkinter
win = tkinter.Tk()  #创建窗口对象
win.title("使用 Entry 组件的简单例子")  #设置窗口标题

e = tkinter.StringVar()
tkinter.Entry(win, textvariable=e).pack()
win.mainloop()
```

这样，在后面的步骤中就可以使用 e.get()获取 Entry 组件的选中内容了，也可以使用 e.set()设置 Entry 组件的内容。

【例 10-18】使用一个 Button 组件获取 Entry 组件的内容：

```
import tkinter
win = tkinter.Tk()  #创建窗口对象
win.title("使用 Entry 组件的简单例子")  #设置窗口标题

def Callbutton():
   print(e.get())

e = tkinter.StringVar()
entry = tkinter.Entry(win, textvariable=e).pack()
```

```
b= tkinter.Button(win, text="获取 Entry 内容", command=Callbutton,
width=10, height=1)  #创建 Button 对象
e.set("Python")
b.pack()
win.mainloop()
```

程序定义了一个 Button 组件和一个 Entry 组件，使用变量 e 绑定 Entry 组件到 Button 组件上。单击 Button 组件会调用 Callbutton()，通过 e.get()函数打印 Entry 组件的状态，效果如图 10-19 所示。

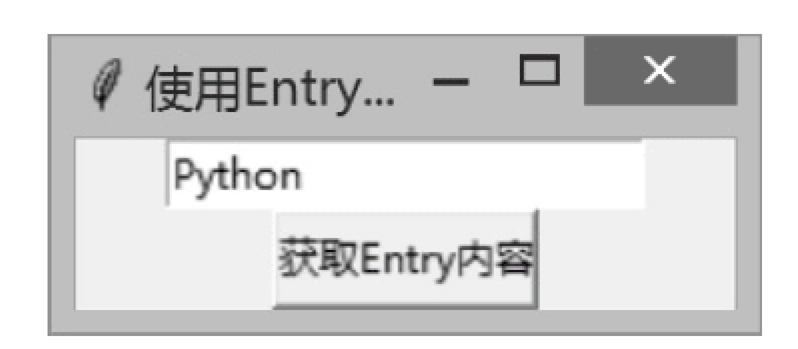


图 10-19　获取 Entry 的内容

10.1.6　单选按钮组件 Radiobutton

Radiobutton 组件用于在窗口中显示单选按钮组件。同一组单选按钮的选项中只可以有一个选项被选中，也就是说，当一个选项被选中后，其他的选项就会自动被取消选中。

创建 Radiobutton 对象的基本方法如下：

```
Radiobutton 对象 = Radiobutton(Tkinter Windows 窗口对象, text = Radiobutton
组件显示的文本内容)
Radiobutton 对象.pack()
```

创建完成后，每个选项会自动成为一个分组。还需要使用 variable 属性为 Radiobutton 组件指定一个变量名。当多个组件被指定同一变量名时，这些组件就会自动地被划归到一个分组，分组后需要使用 value 参数设置每一个选项的值，以表示该值是否被选中。

【例 10-19】使用 Radiobutton 组件的简单例子：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Radiobutton 组件的简单例子")  #设置窗口标题
v = tkinter.IntVar()
v.set(1)
r1 = tkinter.Radiobutton(win, text="男", variable=v, value=1)
  #创建 Radiobutton 组件

r1.pack()  #显示 Radiobutton 组件
r2 = tkinter.Radiobutton(win, text="女", variable=v, value=0)
  #创建 Radiobutton 组件

r2.pack()  #显示 Radiobutton 组件
win.mainloop()
```

运行上面的代码，就会弹出一个如图 10-20 所示的窗口。当选中“男”选项时，

“女”选项被自动取消选中；再次选中“女”选项时，“男”选项被自动取消。



图 10-20　使用 Radiobutton 组件

10.1.7　复选框组件 Checkbutton

Checkbutton 组件用于在窗口中显示复选框。

创建 Checkbutton 的方法如下：

```
Checkbutton 对象 = Checkbutton(Tkinter Windows 窗口对象, text=显示文本内容
command=点击后调用的函数)
```

【例 10-20】使用 Checkbutton 的简单例子：

```
import tkinter
from tkinter import messagebox

def Callcheckbutton():
    messagebox.showinfo(title="", message="提交")

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Checkbutton 组件的简单例子")  #设置窗口标题
b = tkinter.Checkbutton(win, text="Python Tkinter",
command=Callcheckbutton)  #创建 Checkbutton 对象
b.pack()  #显示 Checkbutton 对象
win.mainloop()
```

运行此程序，会显示如图 10-21 所示的窗口，勾选复选框后，会调用 Callcheckbutton()函数，弹出一个如图 10-22 所示的提交消息框。

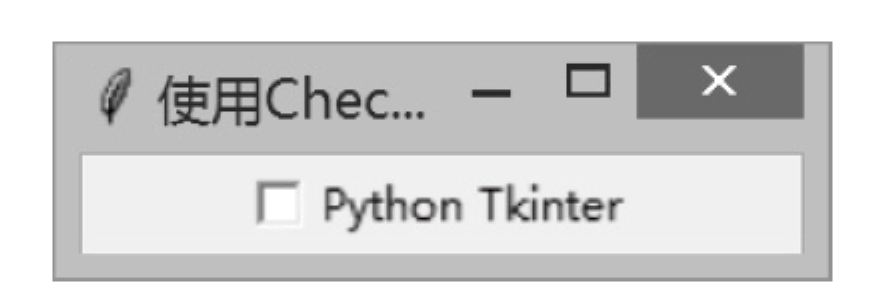


图 10-21　一个 Checkbutton 组件



图 10-22　提交消息框



当然，在实际应用中，我们遇到的更多情况是同时勾选多个选项，然后用一个 Button 按钮进行提交。这时就需要判断 Checkbutton 组件的选项是否被选中。

Checkbutton 组件有 ON 和 OFF 两种状态值，默认状态下 ON 值为 1，OFF 值为 0。也可以使用内置属性 onvalue 来设置 Checkbutton 被选中时的值，使用 offvalue 设置 Checkbutton 组件被取消选中时的值，设置方法如下：

```
Checkbutton(win, text="优秀", onvalue="1", offvalue="0",
command=Callcheckbutton).Pack
win.mainloop()
```

为了获取 Checkbutton 组件的状态，需要使用 variable 属性为 Checkbutton 指定一个变量名，例如：

```
Checkbutton(root, text="优秀", variable=value, onvalue="1", offvalue="0",
command=Callcheckbutton)
```

这样就可以在 Button 按钮中设置被单击后调用 value.get()函数来获取 Checkbutton 组件的被选取状态了。也可以通过使用 value.set()函数来设置 Checkbutton 组件的状态。

【例 10-21】简单的复选框例子：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Checkbutton 组件的简单例子")  #设置窗口标题
v = tkinter.IntVar()

def Callcheckbutton():
   print(v.get())

cb = tkinter.Checkbutton(win, variable=v, text="Checkbutton",
onvalue="1", offvalue="0", command=Callcheckbutton).pack()
b = tkinter.Button(win, text="获取 Checkbutton 状态",
command=Callcheckbutton, width=20).pack()  #创建 Button 组件
v.set("1")
win.mainloop()
```

上面的代码首先定义了一个全选和取消全选的复选框。通过单击调用 checkbutton_on()函数或者 checkbutton_off()函数设置所有的 Checkbutton 组件的状态。再设置一个 Button 按钮，使用 Callcheckbutton()函数将变量 value 绑定到 Button 组件上。单击 Button 按钮后，会调用 Callcheckbutton()函数，打印出 Checkbutton 组件的状态，效果如图 10-23 所示。

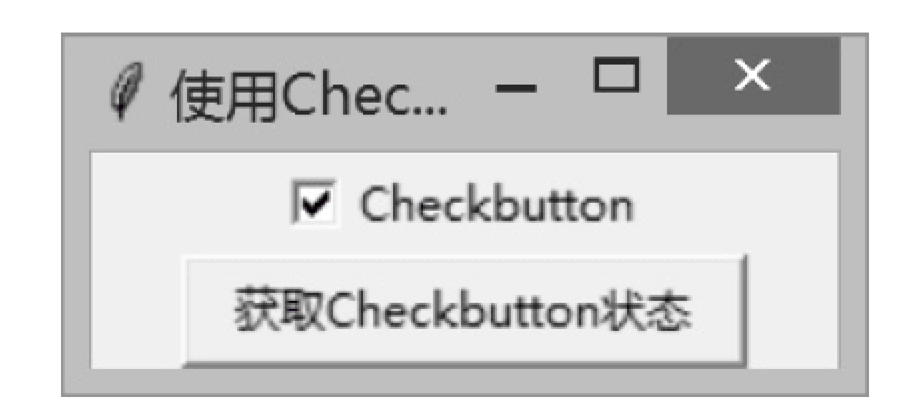


图 10-23　获取复选框的内容

10.1.8　文本框组件 Text

Text 组件用于在窗口中输入多行文本内容。创建 Text 对象的具体方法如下：

```
Text 对象 = Text(Tkinter Windows 窗口对象)
Text 对象.pack()
```

【例 10-22】简单的 Text 组件的例子：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Text 组件的简单例子")  #设置窗口标题
t = tkinter.Text(win)  #创建 Text 组件
t.pack()
win.mainloop()
```

运行上面的程序，就会弹出一个如图 10-24 所示的窗口。



图 10-24　一个 Text 组件

也可以通过 Text.insert()方法向文本框内添加内容。insert()方法的语法如下：

```
Text 组件.insert(插入位置, 插入的文本)
```

其中，插入位置格式为“行数.列数”。

【例 10-23】使用 Text.insert()函数向 Text 组件内添加文本内容：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Text 组件的简单例子")  #设置窗口标题
t = tkinter.Text(win)  #创建 Text 组件
t.insert(1.1, "2017-01-17:12138")
t.insert(1.5, "insterted")
t.pack()
win.mainloop()
```

上面的例子向 Text 组件的第 1 行第 1 列插入数据“2017-01-17:12138”，并在第 1 行第 5 列插入字符“inserted”，运行结果如图 10-25 所示。

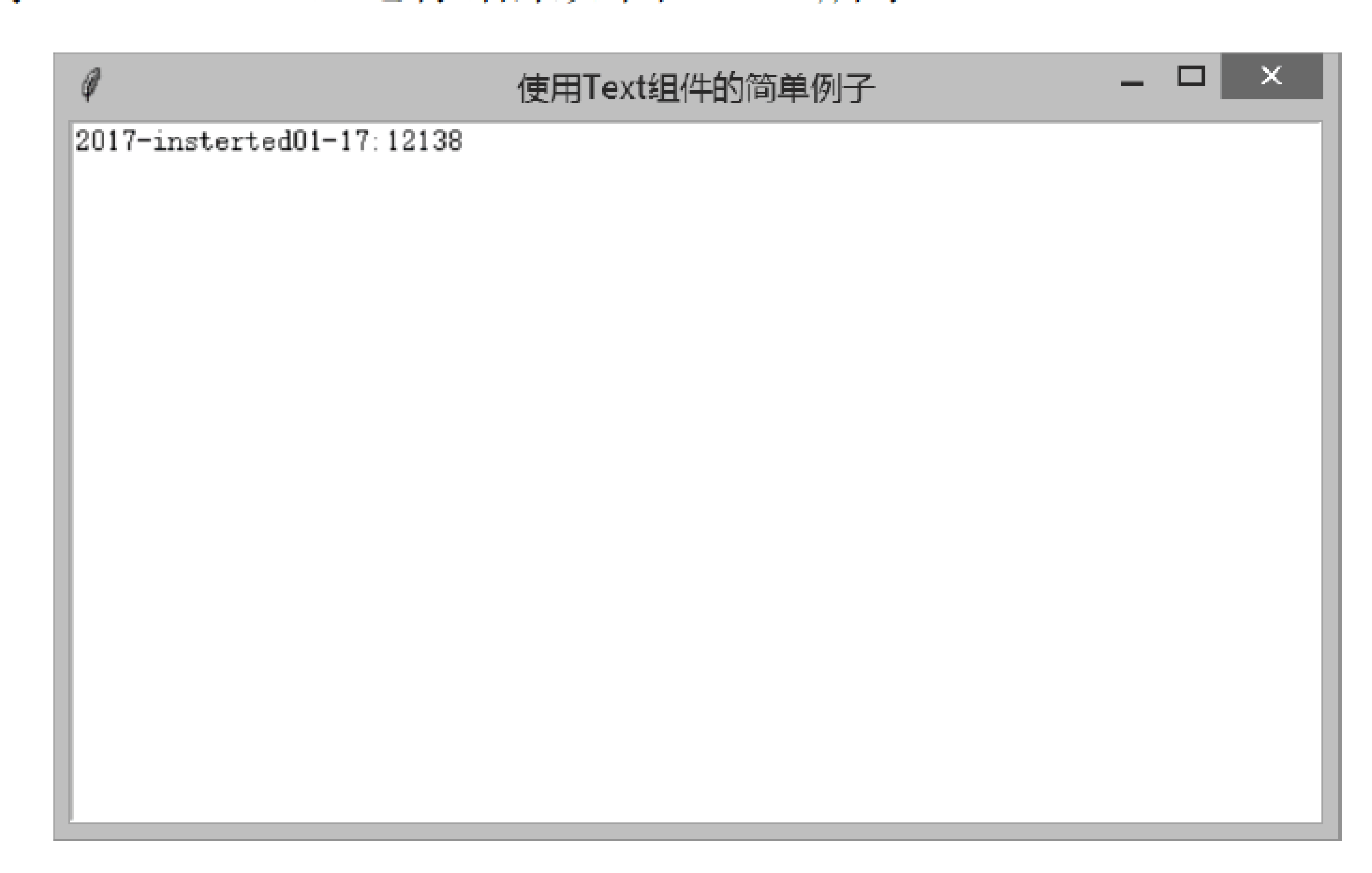


图 10-25　向 Text 添加内容

10.1.9　列表框组件 Listbox

Listbox 组件是一个列表框组件，用于在窗口中显示多个文本项。

1. 创建和显示 Listbox 对象

创建 Listbox 对象的基本方法如下：

```
Listbox 对象 = Listbox(tkinter Windows 窗口对象)
Listbox 对象.pack()
```

可以使用 insert()方法向列表框中添加文本项，方法如下：

```
Listbox 对象.insert(index,item)
```

参数说明如下。

index：插入文本项的位置，如果是在尾部插入文本项，则可以使用 end 参数；如果在当前选中处插入文本项，可以选用 active 选项。

item：插入的文本项。

【例 10-24】使用 Listbox 的简单例子：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Listbox 组件的简单例子")  #设置窗口标题
Lb = tkinter.Listbox(win)
for item in ["北京","上海","天津"]:
   Lb.insert("end",item)
Lb.pack()
win.mainloop()
```

运行上面的程序，会弹出一个如图 10-26 所示的窗口。

2. 设置多选的列表框

将 selectmode 属性设置为 multiple，可以设置多选的列表框。

【例 10-25】设置多选的列表框：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Listbox 组件的简单例子")  #设置窗口标题
Lb = tkinter.Listbox(win,selectmode = "multiple")
for item in ["北京","上海","天津"]:
    Lb.insert("end",item)
Lb.pack()
win.mainloop()
```

运行效果如图 10-27 所示。



图 10-26　一个 Listbox 组件



图 10-27　设置多选的列表框

3. 获取 Listbox 组件的内容

为了获取 Listbox 组件的内容，需要使用 listvariable 属性为 Listbox 组件指定一个对应的变量，例如：

```
L = tkinter.StringVar()
tkinter.Listbox(win, listvariable=L).pack()
win.mainloop()
```

这样一来，在后面的代码中我们就可以用 L.get 方法获取 Listbox 组件的内容了。

【例 10-26】使用一个 Button 按钮组件获取 Listbox 组件的内容：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Listbox 组件的简单例子")  #设置窗口标题
L = tkinter.StringVar()

def Calllistbox():
```

```
    print(L.get())

Lb = tkinter.Listbox(win,listvariable = L)  #创建 Listbox 组件
for item in ["北京","上海","天津"]:
    Lb.insert("end",item)
Lb.pack()  #显示 Listbox 对象
b = tkinter.Button(win, text="获取 Listbox 内容", command=Calllistbox,
width=20)  #创建 Button 组件
b.pack()  #显示 Button 对象
win.mainloop()
```

程序定义了一个 Button 组件和一个 Listbox 组件，并使用变量 L 将 Listbox 组件绑定到 Button 按钮上。单击 Button 按钮，就会调用 Calllistbox()函数，并通过 L.get()函数打印出组件 Listbox 的内容。

10.1.10　菜单组件 Menu

Menu 组件是一个菜单组件，用于在窗口中显示菜单条和下拉菜单。

1. 创建和显示 Menu 对象

创建 Menu 对象的方法如下：

```
Menu 对象 = Menu(Tkinter Windows 窗口对象)
Tkinter Windows 窗口对象["menu"] = menubar
Tkinter Windows 窗口对象.mainloop()
```

可以使用 add_command()方法向 Menu 组件中插入菜单项，方法如下：

```
Menu 对象.add_command(label = 菜单文本, command = 菜单命令函数)
```

【例 10-27】使用 Menu 组件的简单例子：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Menu 组件的简单例子")  #设置窗口标题

def Hello():
    print("这是一个菜单组件")

m = tkinter.Menu(win)
for item in ["文件","编辑","帮助"]:
    m.add command(label = item,command = Hello)
win["menu"] = m
win.mainloop()
```

运行上面的程序，会弹出一个如图 10-28 所示的窗口。

2. 添加下拉菜单

前面介绍的 Menu 组件只是创建了一行主菜单，默认情况下并不包含下拉菜单。我们

可以通过将一个 Menu 组件作为另一个 Menu 下拉菜单的方法实现下拉菜单的添加。具体方法如下：

```
Menu 对象 1.add_command(label = 文本菜单, menu = Menu 对象 2)
```

上面的语句将 Menu 对象 2 设置为 Menu 对象 1 的下拉菜单。注意，在创建 Menu 对象 2 的时候，也要指定它是 Menu 对象 1 的子菜单，方法如下：

```
Menu 对象 2 = menu(Menu 对象 1)
```

【例 10-28】创建带下拉菜单的 Menu 组件：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Menu 组件的简单例子")  #设置窗口标题

def Hello():
   print("这是一个菜单组件")

m = tkinter.Menu(win)
filemenu = tkinter.Menu(m)
for item in ["打开","关闭","退出"]:
   filemenu.add command(label = item,command = Hello)

m.add cascade(label = "文件",menu= filemenu)
m.add cascade(label = "编辑")
m.add cascade(label = "帮助")
win["menu"] = m
win.mainloop()
```

运行结果如图 10-29 所示。



图 10-28　一个 Menu 组件



图 10-29　带下拉菜单的 Menu 组件

10.1.11　滑动条组件 Scale

Scale 组件用于在窗口中以滑块的形式显示一个范围内的数字。可以设置选择数字的最小值、最大值和步长值。

1. 创建和显示 Scale 对象

创建 Scale 对象的基本方法如下：

```
Scale 对象 = Scale(Tkinter Windows 窗口对象, from  = 最小值, to = 最大值,
resolution = 步长值, orient = 显示方向)
Scale.pack()
```

注意： from 是 Python 的关键字，这里添加下划线是为了与关键字 from 区分开，又不失其基本含义。

【例 10-29】使用 Scale 组件的简单例子：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Scale 组件的简单例子")  #设置窗口标题
s = tkinter.Scale(win,
               from  = 0,  #设置最小值
               to = 100,  #设置最大值
               resolution = 1,  #设置步长
               orient = "horizontal"  #设置水平方向
               ).pack()
win.mainloop()
```

运行上面的程序，会显示如图 10-30 所示的窗口。

2. 获取 Scale 组件的值

需要使用 variable 属性为 Scale 组件指定一个变量名，例如：

```
Scale 对象 = scale(win,
from  = 最小值,
to = 最大值,
resolution = 步长值,
orient = 显示方向,
variable = v),pack()
win.mainloop()
```

在后面的代码中，我们就可以使用 v.get()函数来获取 Scale 组件的状态了。也可以使用 v.set()属性来设置 Scale 的值。例如使用下面的语句，可以将上面定义的 Scale 组件的值设置为 50：

```
v.set(50)
```

【例 10-30】使用一个 Button 按钮获取 Scale 组件的值：

```
import tkinter

win = tkinter.Tk()  #创建窗口对象
win.title("使用 Scale 组件的简单例子")  #设置窗口标题
v = tkinter.IntVar()

def Callscale():
    print(v.get())

s = tkinter.Scale(win,
               from  = 0,  #设置最小值
               to = 100,   #设置最大值
               resolution = 1,  #设置步长
               orient = "horizontal",  #设置水平方向
               variable = v
               ).pack()
b = tkinter.Button(win, text="获取 Scale 状态", command=Callscale,
width=20).pack() #创建 Button 组件
v.set(50)
win.mainloop()
```

程序中定义了一个 Button 组件和一个 Scale 组件，使用变量 v 将 Scale 组件绑定到 Button 按钮上。单击 Button 按钮后会调用 Callscale()函数，并通过 v.get()函数打印出 Scale 组件的值，效果如图 10-31 所示。

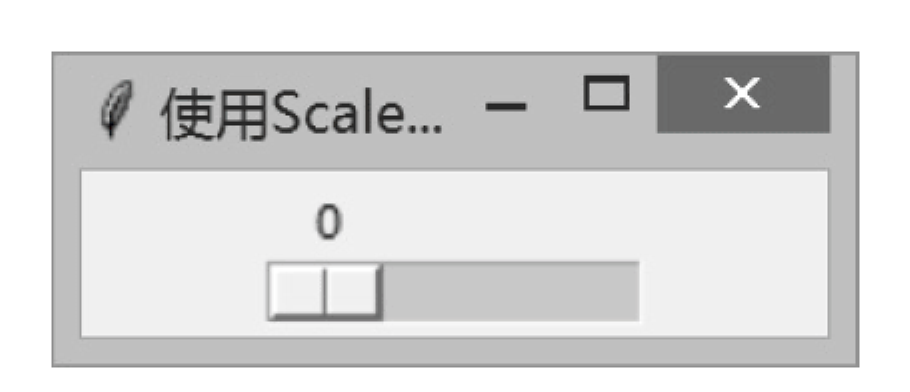


图 10-30　一个 Scale 组件



图 10-31　获取 Scale 的值

10.2　网格布局管理器

上节介绍了大量的 Tkinter 组件，但在实际应用中会发现，各个组件在窗口内的位置并不是很整齐。

有鉴于此，本节引入了网格布局管理器，用于将组件放置到窗体的指定位置。在 Tkinter 模块中有三种布局管理器——grid、pack 和 place。由于 grid 布局管理器应用更加广泛，上手更加容易，所以本节主要介绍 grid 布局管理器。pack 布局管理器相对 grid 管理器灵活度差一些。而 place 布局管理器虽能精确控制组件位置，但编程也相对复杂。针对 Python 快速编程、快速验证的特点，更多的人会选用 grid 布局管理器。

10.2.1　网格

网格是一个假想的矩阵，包含水平线和垂直线，将矩阵分隔成小单元格(cell)。第 1 行单元格的行号为 0，第 2 行单元格的行号为 1，以此类推。同理，第 1 列单元格的列号为 0，第 2 列单元格的列号为 1，以此类推。每一个单元格由行号和列号标识。图 10-32 展示了一个 3×4 的网格和每个单元格的编号。

row 0，column 0	row 0，column 1	row 0，column 2	row 0，column 3
row 1，column 0	row 1，column 1	row 1，column 2	row 1，column 3
row 2，column 0	row 2，column 1	row 2，column 2	row 2，column 3

图 10-32　网格与单元格编号的对应关系

图 10-32 中显示了一个水平和垂直方向均匀分隔的空间，但这并不是常用的 GUI 编程布局，图 10-33 显示了一些常用的布局格式。

图 10-33　常用的布局格式

组件放置到 grid 中可以创建 GUI 界面。组件可以放置在一个单元格中，也可以放置在多个连续的行列中，每个行列会自动扩展以适应最大组件。参数 padx 和 pady 用于具体指定放置到单元格中组件周围空白区域的大小。

默认地，组件会居中显示在单元格中，其属性 sticky 可用于改变组件在单元格中的放置方式，也能控制放大组件使其满足单元格的大小。

图 10-34 显示的是本章一开始所列代码的可视化界面。网格为 5 行 2 列(行号 0 到 4，列号 0 到 1)。每一个标签控件和输入控件都放到一个单元格内。

按钮放置在第 3 行第 0 列位置，占连续两列单元格。窗体中的其他两个组件的声明和布局代码如下：参数 padx ＝ 5，在组件左右两边分别设定 5 个像素的空白边界；参数 pady ＝ 5，在组件上下两边分别设定 5 个像素的空白边界；参数 sticky ＝ W，移动输入框组件到单元格左边，参数 columnspan＝2，设定组件占两个合并单元格。

【例 10-31】房屋贷款计算器程序的 grid 布局代码：

```
from tkinter import *

win = Tk()  #创建窗口对象
win.title("房屋贷款计算器")  #设置窗口标题
```

```
lab entNumber = Label(win,text = "贷款金额:")  #创建 Label 标签
lab entNumber.grid(row=0, column=0, padx=5, pady=5, sticky="e")
  #使用 grid 进行布局

entNumber = Entry(win,width = 15)  #创建 Entry 对象

entNumber.grid(row=0, column=1, padx=5, pady=5, sticky="w")
  #使用 grid 进行布局

lab rate = Label(win,text = "贷款年利率(百分数):")  #创建 Label 标签

lab rate.grid(row=1, column=0, padx=5, pady=5, sticky="e")
  #使用 grid 进行布局

rate = Entry(win,width = 15)  #创建 Entry 对象

rate.grid(row = 1,column = 1,padx = 5,pady = 5,sticky = "w")
  #使用 grid 进行布局

lab years = Label(win,text = "还款年数:")  #创建 Label 标签

lab years.grid(row = 2,column = 0,padx = 5,pady = 5,sticky = "e")
  #使用 grid 进行布局

years = Entry(win,width = 15)  #创建 Entry 对象

years.grid(row = 2,column = 1,padx = 5,pady = 5,sticky = "w")
  #使用 grid 进行布局

bt Calculate = Button(win,text = "计算每月应还款金额:")
  #创建 button 对象，用于提交数据

bt Calculate.grid(row = 3,column = 0,columnspan = 2,pady = 5)
  #使用 grid 进行布局，横跨两列

lab payment = Label(win,text = "每月应还款金额:")  #创建 Label 标签

lab payment.grid(row = 4,column = 0,padx = 5,pady = 5,sticky = "e")
  #使用 grid 进行布局

payment =  Entry(win,width = 15,state = "readonly")  #创建 Entry 对象

payment.grid(row = 4,column = 1,padx = 5,pady = 5,sticky = "w")
  #使用 grid 进行布局

win.mainloop()
```

运行效果如图 10-34 所示。



图 10-34　房屋贷款计算器的布局

一般地，用 widgetName.grid(row = m, column = n)语句放置组件到第 m 行第 n 列的单元格中。grid 方法还包括一些附加属性，如 padx、pady、sticky 等。

使用 widgetName.grid(row=m, colum=n, columnspan=c)语句将组件放在以第 m 行第 n 列为起始位置，连续跨越 c 列的单元格中。如果将 columspan=c 换成 rowspan=c，将连续跨越 c 行。

表 10-4 中的参数用来设定组件空白边界的布局方法。

表 10-4　设定组件空白边界

参　数	效　果
padx = r	在组件左右两边分别加入 r 个空白像素
pady = r	在组件上下两边分别加入 r 个空白像素
padx = (r,s)	在组件左边加入 r 个空白像素，在组件右边加入 s 个空白像素
pady = (r,s)	在组件上边加入 r 个空白像素，在组件下边加入 s 个空白像素

注意： grid 布局管理器的行和列的值并不需要特意指定。grid 布局管理器将自动根据放入 grid 中的组件位置决定其行列。此外，grid 布局管理器每列的宽度和每行的高度都会自动进行调整，以适应其所包含的所有组件的宽度、高度和空白边界。

10.2.2　sticky 属性

设置 sticky 属性的方法如下：

```
widgetName.grid(row = m, column = n, sticky = letter)
```

这里 letter 为 N、S、E 和 W 四个字母之一，它会使空间边缘靠近单元格的顶部、底部、右侧、左侧排列。

【例 10-32】不同 sticky 属性的显示效果：

```
years = Entry(win, width=2)  #创建 Entry 对象
years.grid(row=0, column=1, sticky="e")  #使用 grid 进行布局
```

其中 years = Entry(win, width=2)语句声明输入框组件为 years，years.grid(row=0,

column=1，sticky="e")语句则将输入框组件放置到了 grid 布局管理器中。图 10-35(a)显示了放置在单元格中间的效果，其余几个子图显示了其他几个属性的布局效果。

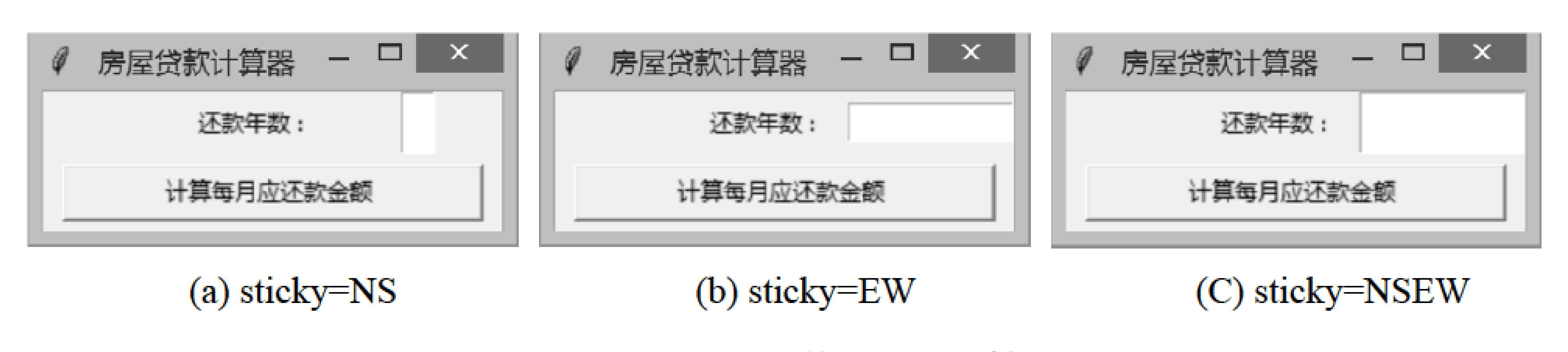


(a) 未设置 sticky　(b) sticky=0

(c) sticky=s　(d) sticky=w　(e) sticky=e

图 10-35　不同的 sticky 属性

sticky 属性的值还可以是 N、S、E 和 W 四个字母的两两组合，或者由四个字母组合在一起。参数 sticky = NS 使得组件的上下边相连，组件被沿垂直方向拉伸；同理，参数 sticky = EW 使得组件沿水平方向被拉伸；参数 NSEW 使得组件沿水平和垂直两个方向被拉伸，以填充整个单元格。上述组合的显示效果如图 10-36 所示。



(a) sticky=NS　(b) sticky=EW　(C) sticky=NSEW

图 10-36　不同的 sticky 属性

10.2.3　向列表框添加垂直滚动条

向列表框添加垂直滚动条，可以通过 Tkinter 的 IstNE 组件和 yscroll 组件来实现。具体代码如下。

【例 10-33】创建滚动条：

```
from tkinter import *

win = Tk()  #创建窗口对象
win.title("创建滚动条的简单例子")  #设置窗口标题

yscroll = Scrollbar(win,orient = VERTICAL)
yscroll.grid(row = 0, column = 2, rowspan = 4, padx = (0,100), pady = 5,
sticky = "ns")

statesList = ["北京","上海","天津","广州","深圳","重庆","杭州"]
conOFlstNE = StringVar()
IstNE = Listbox(win, width = 14, height = 4, listvariable = conOFlstNE,
yscrollcommand = yscroll.set)
```

```
IstNE.grid(row = 0, column = 1, rowspan = 4, padx = (100,0), pady = 5,
sticky = "e")

conOFlstNE.set(tuple(statesList))
yscroll["command"] = IstNE.yview()
win.mainloop()
```

注意：

- 滚动条组件的声明必须在列表框控件之前。
- 参数 yscrollcommand = yscroll.set 必须在列表框组件对象创建的时候添加到其构造器中。
- 代码中必须包括参数 yscroll[“command”] = IstNE.yview()。
- 倒数第二句代码的作用是将滚动条添加到列表框中。

界面的效果如图 10-37 所示，列表框和滚动条组件被放置到相邻的单元格中，用户可以通过单击滚动条上下箭头或者拖曳滚动条组件上的矩形框来选择列表框中的选项。此外，sticky 参数保证了两个组件紧密相连，滚动条垂直填充。参数 padx = (0,100)和 padx = (100,0)用于在列表框左侧和滚动条右侧保留一些边界。

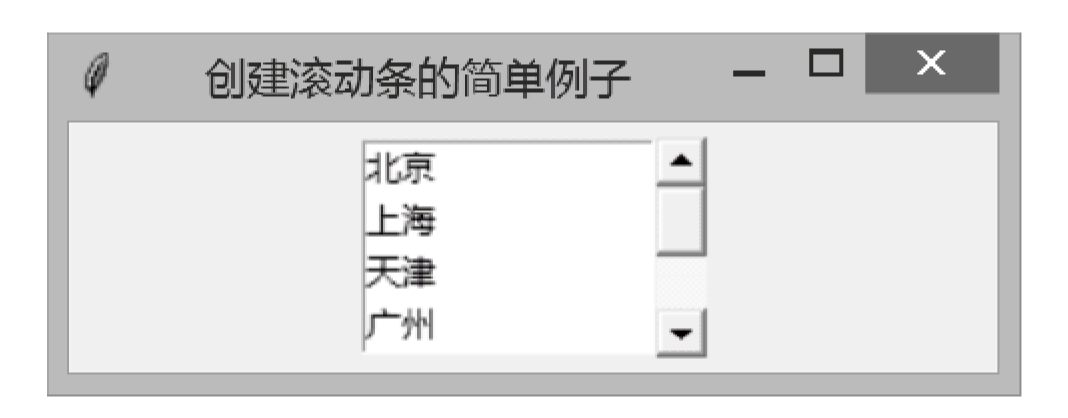


图 10-37　添加垂直滚动条的列表框

10.2.4　设计窗体布局

在 GUI 程序开发中，窗体布局一般遵循如下准则。

(1)　用户通过使用输入框组件键入信息，或者单击列表框中的列表项选择信息。标签组件通常放置在输入框左侧，用于提示输入框组件应该输入什么样的信息；标签组件通常放在列表框上方，用于描述列表框中的内容。

(2)　程序的输出信息通常用只读输入组件或者列表框。如果列表框中显示的内容较多，可以在列表框组件旁边添加垂直滚动条来帮助显示列表框内容。

(3)　一般来讲，按钮横跨多于一列。

(4)　列表框默认包含 10 个列表项。

(5)　在开始界面编程时，建议先在纸上做一个草图设计，观察组件布局，使之更加美观，对于不合理的布局组件，可以适当调整。

(6)　程序第一次运行后，程序员通过增加空白和使用组件中的网格方法的 sticky 参数加以调整，优化设计。这个过程往往要反复多次才能完成。

10.3 GUI 编程

GUI 编程通常采用面向对象的编程方式。然而为了尽可能地简化代码的复杂性，我们采用一种直接的编程方式。在 10.3.2 节将介绍如何用面向对象方式编写 GUI 程序。

10.3.1 将 TUI 程序转换成 GUI 程序

一般来说，程序包含三个部分——输入、加工和输出。

在 TUI 程序中，输入通常用 input 语句从键盘读入数据，或者用 read 等方法从文件读入数据；输出则通常用 print 语句把数据显示到屏幕上，或者用 write 等方法将数据写入到文件中。当将 TUI 程序转换成 GUI 程序时，我们通常会用 Label/Entry 组件来代替 input 和 print 语句，而数据处理部分和文件读写与 GUI 程序相同，主要的区别在于 GUI 程序的处理过程需要由一个事件触发。

下面的例子将本章开始时图 10-1(a)所示的 TUI 程序转化成 GUI 程序，网格由 5 行 2 列组成。

【例 10-34】使用 GUI 界面的房贷计算器：

```
from tkinter import *
from tkinter import messagebox

win = Tk()  #创建窗口对象
win.title("使用 GUI 界面的简单房贷计算器")  #设置窗口标题

def Calculation():
    try:
        Principal = float(number of ent.get())
        year rate = float(number of rate.get())
        month rate = year rate / 12  #转换成月利率
        years = float(number of years.get())
        months = years * 12  #转换成月数
        sum = (Principal * (month rate/100) * pow((1 + month rate/100),
months))\
              /(pow((1 + month rate/100), months)-1)  #月还款计算公式
        number of repay.set(round(sum, 2))  #保留两位小数并输出
    except:
        messagebox.showerror(title="提示", message="输入错误，请重新输入")

lab entNumber = Label(win, text="贷款金额:")  #创建 Label 标签
lab entNumber.grid(row=0, column=0, padx=5, pady=5, sticky="e")
  #使用 grid 进行布局

number of ent = IntVar()
entNumber = Entry(win, width=15, textvariable=number of ent)
  #创建 Entry 对象
```

```
entNumber.grid(row=0, column=1, padx=5, pady=5, sticky="w")
  #使用 grid 进行布局

lab rate = Label(win, text="贷款年利率(百分数):")  #创建 Label 标签
lab rate.grid(row=1, column=0, padx=5, pady=5, sticky="e")
  #使用 grid 进行布局
number of rate = DoubleVar()
rate = Entry(win, width=15, textvariable=number of rate)  #创建 Entry 对象
rate.grid(row=1, column=1, padx=5, pady=5, sticky="w")  #使用 grid 进行布局

lab years = Label(win, text="还款年数:")  #创建 Label 标签
lab years.grid(row=2, column=0, padx=5, pady=5, sticky="e")
  #使用 grid 进行布局
number of years = IntVar()
years = Entry(win, width=15, textvariable=number of years)
  #创建 Entry 对象
years.grid(row=2, column=1, padx=5, pady=5, sticky="w")  #使用 grid 进行布局

bt Calculate = Button(win, text="计算每月应还款金额", command=Calculation)
  #创建 button 对象，用于提交数据
bt Calculate.grid(row=3, column=0, columnspan=2, pady=5)
  #使用 grid 进行布局，横跨两列

lab payment = Label(win, text="每月应还款金额:")  #创建 Label 标签
lab payment.grid(row=4, column=0, padx=5, pady=5, sticky="e")
  #使用 grid 进行布局
number of repay = DoubleVar()
payment = Entry(win, width=15,
state="readonly",textvariable=number of repay)  #创建 Entry 对象
payment.grid(row=4, column=1, padx=5, pady=5, sticky="w")
  #使用 grid 进行布局
win.mainloop()
```

运行后的界面如图 10-1(b)所示。通过比较图 10-1(a)与图 10-1(b)，读者不难想象，无论是从视觉效果来看，还是从用户操作的方便性来看，图形用户界面远优于文本界面，这也是 GUI 非常流行的根本原因。

10.3.2　面向对象编程

关于面向对象的方法，第 6 章已做详细介绍，本章不复赘述。下面的例子就是使用面向对象方法编写的 GUI 房贷计算器。

【例 10-35】使用面向对象方式编写的 GUI 房贷计算器：

```
from tkinter import *
from tkinter import messagebox

class MortgageCalculator:

    def __init__(self):
```

```
        win = Tk()  #创建窗口对象
        win.title("使用 GUI 界面的简单房贷计算器")  #设置窗口标题
        lab entNumber = Label(win, text="贷款金额:")  #创建 Label 标签
        lab entNumber.grid(row=0, column=0, padx=5, pady=5, sticky="e")
           #使用 grid 进行布局
        self.number of ent = IntVar()
        entNumber = Entry(win, width=15, textvariable=self.number of ent)
           #创建 Entry 对象
        entNumber.grid(row=0, column=1, padx=5, pady=5, sticky="w")
           #使用 grid 进行布局

        lab rate = Label(win, text="贷款年利率(百分数):")  #创建 Label 标签
        lab rate.grid(row=1, column=0, padx=5, pady=5, sticky="e")
           #使用 grid 进行布局
        self.number of rate = DoubleVar()
        rate = Entry(win, width=15, textvariable=self.number of rate)
           #创建 Entry 对象
        rate.grid(row=1, column=1, padx=5, pady=5, sticky="w")
           #使用 grid 进行布局

        lab years = Label(win, text="还款年数:")  #创建 Label 标签
        lab years.grid(row=2, column=0, padx=5, pady=5, sticky="e")
           #使用 grid 进行布局
        self.number of years = IntVar()
        years = Entry(win, width=15, textvariable=self.number of years)
           #创建 Entry 对象
        years.grid(row=2, column=1, padx=5, pady=5, sticky="w")
           #使用 grid 进行布局

        bt Calculate = Button(win, text="计算每月应还款金额",
          command=self.Calculation)  #创建 button 对象，用于提交数据
        bt Calculate.grid(row=3, column=0, columnspan=2, pady=5)
          #使用 grid 进行布局，横跨两列

        lab payment = Label(win, text="每月应还款金额:")  #创建 Label 标签
        lab payment.grid(row=4, column=0, padx=5, pady=5, sticky="e")
           #使用 grid 进行布局
        self.number of repay = DoubleVar()
        payment = Entry(win, width=15, state="readonly",
           textvariable=self.number of repay)  #创建 Entry 对象
        payment.grid(row=4, column=1, padx=5, pady=5, sticky="w")
           #使用 grid 进行布局
        win.mainloop()

    def Calculation(self):
        try:
            Principal = float(self.number of ent.get())
            year rate = float(self.number of rate.get())
            month rate = year rate / 12  #转换成月利率
            years = float(self.number_of_years.get())
```

```
            months = years * 12  #转换成月数
            sum = (Principal * (month rate/100) * pow((1 + month rate/100),
months))\
                  /(pow((1 + month rate/100), months)-1)  #月还款计算公式
            self.number of repay.set(round(sum, 2))  #保留两位小数并输出
        except:
            messagebox.showerror(title="提示", message="输入错误，请重新输入")

MortgageCalculator()
```

运行后的界面如图 10-1(b)所示。

10.4　案例实训：设计一个查看文件目录的程序

本节案例将结合前面几章所学内容，并结合 GUI 编程，以面向对象的方法设计一个查看文件目录的程序。该程序完成的功能如下。

(1) 默认显示当前目录下的所有目录和文件。

(2) 双击某目录，能够显示该目录下的所有目录和文件。

(3) 可以通过滚动条查看不在可视区域内的文件。

(4) 按钮 Clear：清除列表框中的所有内容。

(5) 按钮 LS：配合路径输入文本框，判断路径是否合法，如果合法则列出所有文件。

(6) 按钮 Quit：结束运行。

查看文件目录程序的具体代码如下：

```
import os
from time import sleep
from tkinter import *

class DirList(object):

    def   init   (self,initdir=None):
        self.top=Tk()
        self.top.title("目录查看器")
        self.title label=Label(self.top,text="目录查看器 1.0 版")
        self.title label.pack()
        self.cwd=StringVar(self.top)
        self.cwd lable=Label(self.top,fg='Red',font=('宋体',10,'bold'))
        self.cwd lable.pack()
        self.dirs frame=Frame(self.top)
        self.sbar=Scrollbar(self.dirs frame)
        self.sbar.pack(side=RIGHT,fill=Y)
        self.dirs listbox=Listbox(self.dirs frame,height=15,width=50,
          yscrollcommand=self.sbar.set)
        self.dirs listbox.bind('<Double-1>',self.setDirAndGo)
        self.dirs listbox.pack(side=LEFT,fill=BOTH)
        self.dirs frame.pack()

        self.dirn=Entry(self.top,width=50,textvariable=self.cwd)
```

```
        self.dirn.bind("<Return>",self.doLS)
        self.dirn.pack()

        self.bottom frame=Frame(self.top)
        self.ret=Button(self.bottom frame,text="Return",
          command=self.RetDir,activeforeground='white',
          activebackground='Gray')
        self.clr=Button(self.bottom frame,text="Clear",
          command=self.clrDir,activeforeground='white',
          activebackground='Gray')
        self.ls=Button(self.bottom frame,text="List",
          command=self.doLS,activeforeground='white',
          activebackground='Gray')
        self.quit=Button(self.bottom frame,text="Quit",
          command=self.top.quit,activeforeground='white',
          activebackground='Gray')
        self.clr.pack(side=LEFT)
        self.ret.pack(side=LEFT)
        self.ls.pack(side=LEFT)
        self.quit.pack(side=LEFT)
        self.bottom frame.pack()

        if initdir:
            self.cwd.set(os.curdir)
            self.doLS()

    def RetDir(self,ev=None):
        dirlist=os.listdir(root)
        dirlist.sort()
        os.chdir(root)
        self.dirs listbox.delete(0,END)
        self.dirs listbox.insert(END,os.curdir)
        self.dirs listbox.insert(END,os.pardir)
        for eachfile in dirlist:
            self.dirs listbox.insert(END,eachfile)
        self.cwd.set(os.curdir)
        self.dirs listbox.config(selectbackground='Gray')


    def clrDir(self,ev=None):
        #self.cwd.set('')
        self.dirs listbox.delete(first=0,last=END)

    def setDirAndGo(self, ev=None):
        self.last=self.cwd.get()
        self.dirs listbox.config(selectbackground='red')
        try:
            check=self.dirs listbox.get(self.dirs listbox.curselection())
        except:
            check=""
        if not check:
                check=os.curdir
        #check is the selected item for file or directory
```

```
        self.cwd.set(check)
        self.doLS()

    def doLS(self,ev=None):
        error=""
        self.cwd lable.config(text=self.cwd.get())
        tdir=self.cwd.get()#get the current working directory
        if not tdir:
            tdir=os.curdir
        if not os.path.exists(tdir):
            error=tdir+':未找到该文件'
        elif not os.path.isdir(tdir):
            error=tdir+":该文件不是文件夹"
        if error:#if error occured
            print(error)
            self.cwd.set(error)
            self.top.update()
            sleep(2)
            if not (hasattr(self,'last') and self.last):
                self.last=os.curdir
                self.cwd.set(self.last)
                self.dirs listbox.config(selectbackground='Gray')
                self.top.update()
                return
        self.cwd.set("目录加载中..")
        self.top.update()
        try:
            dirlist=os.listdir(tdir)
            dirlist.sort()
            os.chdir(tdir)
        except:
            tdir = "."
            dirlist=os.listdir(tdir)
            dirlist.sort()
            os.chdir(tdir)
        self.dirs listbox.delete(0,END)
        self.dirs listbox.insert(END,os.curdir)
        self.dirs listbox.insert(END,os.pardir)
        for eachfile in dirlist:
            self.dirs listbox.insert(END,eachfile)
        self.cwd.set(os.curdir)
        self.dirs listbox.config(selectbackground="Gray")

def main():
    global root
    root = os.getcwd()
    print(root)
    d=DirList(root)
    mainloop()

if   name   == '  main  ':
    main()
```

运行上面的代码后，显示的程序界面如图 10-38 所示。



图 10-38　程序运行界面

本 章 小 结

在本章中，介绍了如何使用 Python 语言进行 GUI 编程，介绍了 Tkinter 模块的 Label 组件、Button 组件、Messagebox 组件、Entry 组件、Radiobutton 组件、Checkbutton 组件、Text 组件、Listbox 组件、Menu 组件、Scale 组件等，并讲解了如何使用网格布局管理器让这些组件更好地展现在窗体中。

在实际的应用中，一个有着良好界面的程序能提供更好的用户体验。合理地运用各个组件，合理地进行布局设计，才能实现多种多样的用户界面。

习　　题

1. 填空题

(1) Python 的常用 GUI 工具有(　　　)、(　　　)、(　　　)和(　　　)等。

(2) Python 图形用户界面程序一般包含一个顶层窗口，也称为(　　　)或(　　　)。

(3) Tkinter 提供了三种不同的几何布局管理器：(　　　)、(　　　)和(　　　)，用于将组件放置到窗体的指定位置，从而组织和管理子配件在父组件中的布局方式。

(4) 通过组件的(　　　)和(　　　)属性，可以设置组件的宽度和高度。

(5) 通过 Button 组件的(　　　)属性可以设置其显示的位图，自定义位图为(　　　)格式的文件。

(6) (　　　)控件用于选择同一组单选按钮中的一个单选按钮(不能同时选择多个)，按钮上可显示文本，也可显示图像。

(7) (　　　)用于显示对象列表，并允许用户选择一个或多个项。

(8) Tkinter 模块中的(　　　)通常用于实现通用消息框的功能，(　　　)用于实现列表框的功能，(　　　)用于实现文本框的功能。

2. 选择题

(1) 在 Tkinter 中，下面的(　　)语句用来创建只读文本框。

A. messagebox　　B. Label　　C. Entry　　D. Text

(2) 弹出消息框是图形界面的一个基本功能，使用 tkinter.messagebox 模块的 askretrycancel()可以弹出一个包含(　　)的疑问消息框。

A. “是”和“否”　　B. “重试”和“取消”

C. “确定”和“取消”　　D. “重置”和“取消”

(3) 使用 grid 布局管理器的 sticky 属性对组件进行调整时，如果想让组件沿水平和垂直两个方向被拉伸，以填充整个单元格，应该将 sticky 的值设定为(　　)。

A. ns　　B. we　　C. center　　D. nsew

3. 问答题

(1) grid 是 Tkinter 模块中的三种布局管理器之一，其中 sticky 属性的值可取 N、S、W、E 之一或其组合，你怎样理解其含义？

(2) 需要将窗口 win 的尺寸设置为 1024×768，最小 800×600，最大 1440×900，应怎样设置？

4. 实验操作题

应用面向对象的编程方法创建一个窗口(300×120)，窗口中布置两个按钮，其中一个按钮用于关闭窗口，另一个按钮用于切换执行传入的两个函数 sta()和 sto()。部分代码已经给出，请补充类 Window 的代码:

```
import time
import tkinter as tk

class Window:

___________________________


def sta():
   print('start.')
   return True
def sto():
   print('stop.')
   return True

if __name__ == '__main__':
   import sys, os

   w = Window(staFunc=sta, stoFunc=sto)
   w.staIco = os.path.join(sys.exec_prefix, 'DLLs\pyc.ico')
   w.stoIco = os.path.join(sys.exec_prefix, 'DLLs\py.ico')
   w.loop()
```

2. 选择题

(1) 在 Tkinter 中，下面的(　　)语句用来创建只读文本框。

A. messagebox　　B. Label　　C. Entry　　D. Text

(2) 弹出消息框是图形界面的一个基本功能，使用 tkinter.messagebox 模块的 askretrycancel()可以弹出一个包含(　　)的疑问消息框。

A. “是”和“否”　　B. “重试”和“取消”

C. “确定”和“取消”　　D. “重置”和“取消”

(3) 使用 grid 布局管理器的 sticky 属性对组件进行调整时，如果想让组件沿水平和垂直两个方向被拉伸，以填充整个单元格，应该将 sticky 的值设定为(　　)。

A. ns　　B. we　　C. center　　D. nsew

3. 问答题

(1) grid 是 Tkinter 模块中的三种布局管理器之一，其中 sticky 属性的值可取 N、S、W、E 之一或其组合，你怎样理解其含义？

(2) 需要将窗口 win 的尺寸设置为 1024×768，最小 800×600，最大 1440×900，应怎样设置？

4. 实验操作题

应用面向对象的编程方法创建一个窗口(300×120)，窗口中布置两个按钮，其中一个按钮用于关闭窗口，另一个按钮用于切换执行传入的两个函数 sta()和 sto()。部分代码已经给出，请补充类 Window 的代码:

```
import time
import tkinter as tk

class Window:

___________________________


def sta():
   print('start.')
   return True
def sto():
   print('stop.')
   return True

if __name__ == '__main__':
   import sys, os

   w = Window(staFunc=sta, stoFunc=sto)
   w.staIco = os.path.join(sys.exec_prefix, 'DLLs\pyc.ico')
   w.stoIco = os.path.join(sys.exec_prefix, 'DLLs\py.ico')
   w.loop()
```



第 11 章

多进程与多线程

本章要点

(1) 多进程与多线程的概念。
(2) 多进程与多线程的区别。
(3) 进程间通信技术。
(4) 进程池。
(5) thread 锁。

学习目标

(1) 了解多进程与多线程的主要区别。
(2) 掌握多进程编程方法。
(3) 掌握多线程编程方法。
(4) 掌握进程间通信技术 queue 消息队列和 pip 管道的应用方法。
(5) 掌握进程池的使用方法。
(6) 掌握 thread 锁的相关知识。

本章主要介绍如何使用 Python 实现程序的并发运行。并发可分为多线程和多进程两种，本章将对这两种技术的区别和使用方法做详细介绍。

11.1 多进程与多线程

11.1.1 为何需要多进程(或多线程)/为何需要并发

随着计算机技术的不断发展，多核计算机已经走入每个人的工作和生活中。在购买计算机时，四核心八线程、八核心十六线程等已经成为评价一台计算机优劣的标准。很多人只是知道，核心越多，程序运行的速度越快，因为核心越多，能同时运行的程序就越多。其实，这背后运用的就是多任务(multitask)技术。

在同一个时间里，同一个计算机系统中如果允许两个或两个以上的进程处于运行状态，这便是多任务。多任务带来的好处是明显的，比如用户可以边听歌、边上网、边打印，而这些任务之间丝毫不会相互干扰。使用多进程(或者多线程)技术，可以大大地提高计算机的运行效率。

11.1.2 多进程与多线程的区别

进程(process)是程序在计算机上的一次执行活动。当运行一个程序时，实质上就启动了一个进程。显然，程序是死的(静态的)，进程是活的(动态的)。进程可以分为系统进程和用户进程。线程(thread)是程序中一个单一的顺序控制流程。进程是一个相对独立的、可调度的执行单元，是系统独立调度和分派 CPU 的基本单位(指运行中程序的调度单位)。进程是资源分配的最小单位，线程是 CPU 调度的最小单位。

我们通常使用的计算机中只有一个 CPU，也就是说只有一颗“心”，要让它一“心”多用，同时运行多个进程，就必须使用并发技术。实现并发技术相当复杂，最容易理解的

是“时间片轮转进程调度算法”。其基本思想可以简要地描述如下：在操作系统的管理下，所有正在运行的进程轮流使用 CPU，每个进程允许占用 CPU 的时间非常短(比如 10 毫秒)，这样，用户根本感觉不到 CPU 是在轮流为多个进程服务，就好像所有的进程都在不间断地运行一样。但实际上，在任何一个时刻，有且仅有一个进程占用 CPU。

如果一台计算机有多个 CPU，情况就不同了，如果进程数小于或等于 CPU 数，则不同的进程可以分配给不同的 CPU 来运行。这样，多个进程就是真正同时运行的，这便是并行。但如果进程数大于 CPU 数，则仍然需要使用并发技术。在 Windows 中，进行 CPU 分配是以线程为单位的，一个进程可能由多个线程组成，这时情况更加复杂，但简单地说有如下关系：

- 总线程数≤CPU 数量：并行运行。
- 总线程数＞CPU 数量：并发运行。

并行运行的效率显然高于并发运行。所以在多 CPU 的计算机中使用多任务技术，可以明显地提高运行效率。

那么，在实际应用中，到底是该用多线程还是该用多进程呢？答案是，好坏是相对的，需要根据实际情况进行选择。表 11-1 根据不同的应用方向，对多进程和多线程进行了简略的对比。

表 11-1　多进程和多线程对比

对比方面	多 进 程	多 线 程	总　结
数据共享、同步	数据共享复杂，需要用 IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也是因为这个原因，导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU 利用率低	占用内存少，切换简单，CPU 利用率高	线程占优
创建、销毁、切换	创建、销毁、切换复杂，速度慢	创建、销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布式；一台机器不够时，扩展到多台机器比较简单	适应于多核分布式	进程占优

从表 11-1 中可以看出如下几点。

(1) 需要频繁创建、销毁的优先用线程。这种原则最常见的应用就是 Web 服务器了，来一个连接建立一个线程，断了就销毁线程。要是用进程，创建和销毁的代价是很难承受的。

(2) 需要进行大量计算的优先使用线程。大量计算要耗费很多 CPU 时间，而且需要频繁切换，这种情况下线程是最合适的。这种原则最常见的应用是图像处理、算法处理。

(3) 强相关的处理用线程，弱相关的处理用进程。什么叫强相关、弱相关？理论上很难定义，给个简单的例子就明白了。一般的 Server 需要完成如下任务：消息收发、消息处

理。“消息收发”和“消息处理”就是弱相关的任务，而“消息处理”里面可能又分为“消息解码”、“业务处理”，这两个任务相对来说，相关性就要强多了。因此“消息收发”和“消息处理”可以分进程设计，“消息解码”、“业务处理”可以分线程设计。当然这种划分方式不是一成不变的，也可以根据实际情况进行调整。

(4) 可能要扩展到多机分布的用进程，多核分布的用线程。

(5) 都满足需求的情况下，用你最熟悉、最拿手的方式。至于“数据共享、同步”、“编程、调试”、“可靠性”这几个维度的所谓的“复杂”、“简单”应该怎么取舍呢？只能说：没有明确的选择方法，选择原则是：如果多进程和多线程都能够满足要求，那么选择你最熟悉、最拿手的那个。

需要提醒的是：虽然给出了这么多的选择原则，但实际应用中基本上都是“进程+线程”的结合方式，千万不要陷入一种非此即彼的误区。

11.2 多进程编程

11.2.1 进程的概念

进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

进程的概念主要有两点：第一，进程是一个实体，每一个进程都有它自己的地址空间，一般情况下，包括文本区域(text region)、数据区域(data region)和堆栈区域(stack region)，文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时(操作系统执行之)，它才能成为一个活动的实体，我们称其为进程。

11.2.2 进程的特征

一个进程具有如下特征。

- 动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生、动态消亡的。
- 并发性：任何进程都可以同其他进程一起并发执行。
- 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。
- 异步性：进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。
- 结构特征：进程由程序、数据和进程控制块三部分组成。

多个不同的进程可以包含相同的程序：一个程序在不同的数据集里就构成不同的进

程，能得到不同的结果，但是执行过程中，程序不能发生改变。

11.2.3　进程的状态

前面我们讲过，在实际运行中，多个进程轮流使用 CPU。进程执行时的间断性，决定了进程可能具有多种状态。事实上，运行中的进程可能具有以下三种基本状态。

(1)　就绪状态(Ready)。

进程已获得除处理器外的所需资源，等待分配处理器资源，只要分配了处理器进程就可执行。就绪进程可以按多个优先级来划分队列。例如，当一个进程由于时间片用完而进入就绪状态时，排入低优先级队列；当进程由 I/O 操作完成而进入就绪状态时，排入高优先级队列。

(2)　运行状态(Running)。

进程占用处理器资源；处于此状态的进程的数目小于或等于处理器的数目。在没有其他进程可以执行时(如所有进程都在阻塞状态)，通常会自动执行系统的空闲进程。

(3)　阻塞状态(Blocked)。

由于进程等待某种条件(如 I/O 操作或进程同步)，在条件满足之前无法继续执行。该事件发生前，即使把处理器资源分配给该进程，也无法运行。

进程间的状态切换如图 11-1 所示。

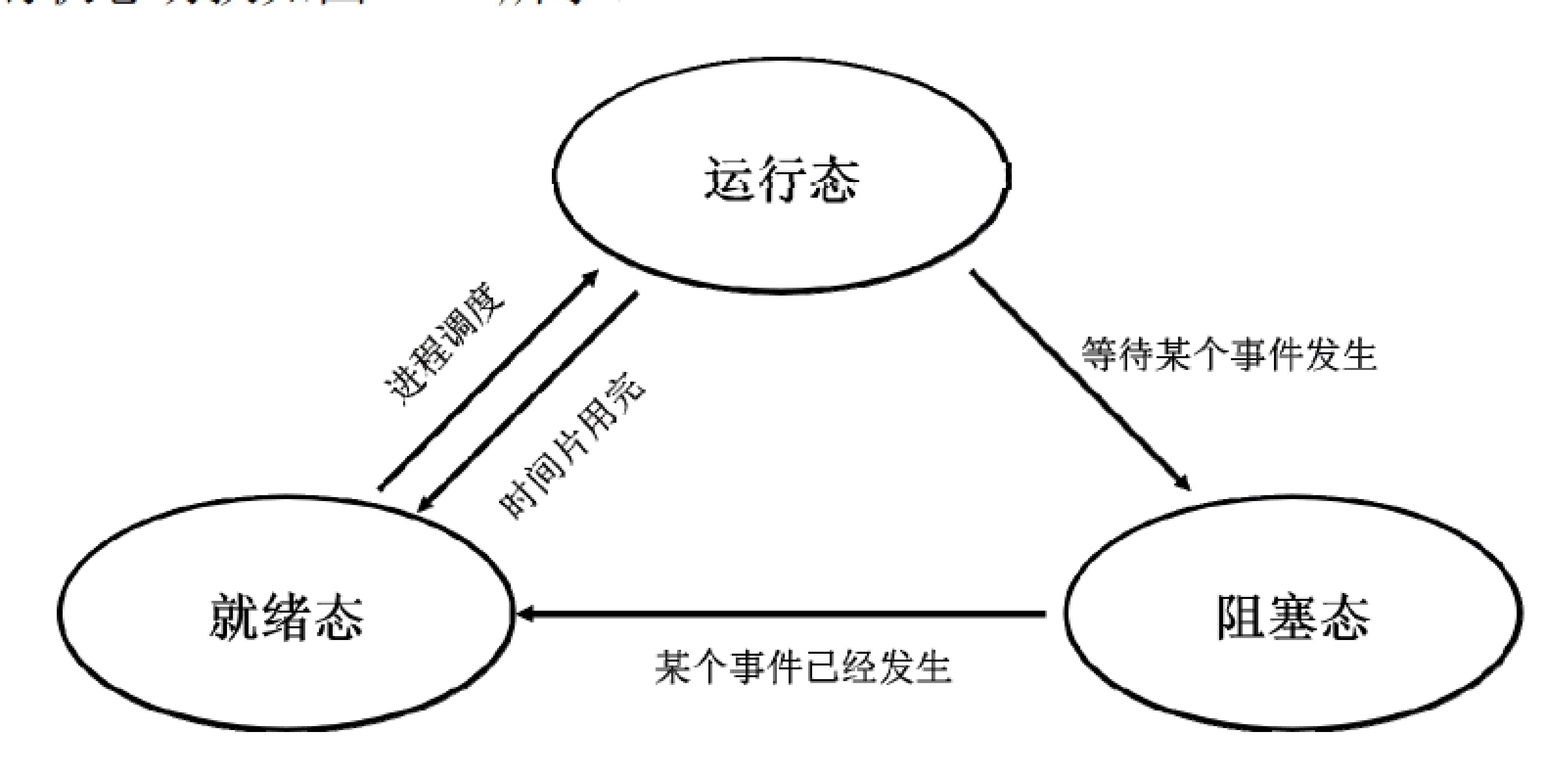


图 11-1　进程间的状态切换

进程间状态切换的控制流程要点如下。

(1)　程序只有作为进程时才能在系统中运行。因此，为使程序能运行，就必须为它创建进程。一旦操作系统发现了要求创建新进程的事件后，便调用进程创建原语 create()创建一个新进程。如果进程就绪队列能够接纳新进程，便将新进程插入到就绪队列中，这时的进程处于就绪状态。

(2)　当处理器空闲时，就会开始运行该进程的指令，这时的程序处于“运行状态”。

(3)　当正在执行的进程请求操作系统提供服务时，由于某种原因(请求系统服务、启动某种操作、新数据尚未到达、无新工作可做等)，操作系统不能立即满足该进程的要求时，该进程只能转变为阻塞状态来等待，一旦要求得到满足后，进程就被唤醒。

(4)　当被阻塞的进程所期待的事件出现时(如 I/O 完成或者其所期待的数据已经到达)，则由有关进程首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其状态由“阻

塞”改为“就绪”，然后再将其插入到就绪队列中。

(5) 当进程顺利运行完成后或者由于异常等原因被操作系统终止时，该进程就会被从内存中移除或者标为“被终止”状态，系统回收全部资源，返回给父进程或者操作系统。

11.3 Multiprocessing

Multiprocessing(多进程)是 Python 提供的非常好用的多进程包，用户只需要简单地定义一个函数，Python 就会自动地完成其他的所有事情。Multiprocessing 支持子进程、进程间通信、数据共享、不同形式的同步，提供了包括 Process、Queue、Pipe、Lock 在内的各种组件。合理运用这些组件，可以轻松地完成单进程到并发执行的转换。

11.3.1 创建进程 Process 模块

创建进程的类为：

```
class multiprocessing.Process(group=None, target=None, name=None,
args=(), kwargs={})
```

其中 group 为 None，它的存在是为了兼容 threading.Thread，target 表示调用对象，name 为别名，args 表示调用对象的位置参数元组，kwargs 表示调用对象的字典。

【例 11-1】创建函数并将其作为单个进程：

```
import multiprocessing
import time

def worker 1(interval):
    print("worker 1 start work")
    time.sleep(interval)
    print("worker 1 end")

if   name   == "  main  ":
    p = multiprocessing.Process(target = worker 1, args = (3,))
    p.start()
    print("p.pid:", p.pid)     #p.pid 为当前进程的标识符，以整数表示
    print ("p.name:", p.name)
      #p.name 为当前进程的名称，以 Process-n 表示，其中 n 为整数
    print ("p.is alive:", p.is alive())
      #p.is_alive 方法测试进程是否处于运行状态，结果为 True 或 False
```

输出结果：

```
p.pid: 13336
p.name: Process-1
p.is_alive: True
worker_1 start work
worker_1 end
```

注意： 由于 IDLE 不能很好地支持多进程输出，所以本例及以下各例的运行环境均为 PyCharm。

在上面的例子中，定义了一个 worker_1 函数，然后在主函数中通过 Process 创建进程并调用。start()方法开始运行，其中 p.pid 为查看当前进程的进程标识符，p.name 为查看当前进程名称，p.is_alive 方法测试进程是否处于运行状态。如果进程还没有执行完，则会返回 True，如果进程已经执行完，则返回 False。

【例 11-2】创建函数并将其作为多个进程：

```
import multiprocessing
import time

def worker_1(interval):
    print ("worker 1")
    time.sleep(interval)
    print ("end worker 1")

def worker_2(interval):
    print ("worker 2")
    time.sleep(interval)
    print ("end worker 2")

def worker_3(interval):
    print ("worker 3")
    time.sleep(interval)
    print ("end worker 3")

if __name__ == "__main__":
    p1 = multiprocessing.Process(target = worker_1, args = (2,))
    p2 = multiprocessing.Process(target = worker_2, args = (3,))
    p3 = multiprocessing.Process(target = worker_3, args = (4,))

    p1.start()
    p2.start()
    p3.start()

    print("The number of CPU is:" + str(multiprocessing.cpu_count()))
    for p in multiprocessing.active_children():
        print("child   p.name:" + p.name + "\tp.id" + str(p.pid))
```

输出结果：

```
The number of CPU is:4
child   p.name:Process-2    p.id10148
child   p.name:Process-3    p.id5224
child   p.name:Process-1    p.id7292
worker_2
worker_1
worker_3
end worker_1
end worker_2
end worker_3
```

上述代码中使用 multiprocessing.cpu_count()函数查看当前计算机的 CPU 数量，并通过语句 multiprocessing.active_children()和 for 循环的结合使用，帮助我们清晰地查看当前活动的进程数。当然，也可以利用 Process 创建多个进程 p1、p2、p3，再通过 start()运行。不难看出，运行结果以 worker_2、worker_1、worker_3 的顺序输出，说明这三个进程间是并发执行的，所以每次开始时的输出顺序可能有所不同。

11.3.2 守护进程 Daemon

守护进程(Daemon)是一种运行在后台的特殊进程，它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。在 Linux 中，系统与用户进行交流的每个界面都称为终端，从某终端开始运行的每一个进程都会依附于这个终端，因此，这个终端被称为这些进程的控制终端，当控制终端被关闭的时候，相应的进程都会自动关闭。但是守护进程却能突破这种限制，它脱离于终端而在后台运行。它脱离终端的目的是为了避免进程在运行过程中的信息在任何终端上显示，并且进程也不会被任何终端所产生的终端信息所打断。它从被执行的时候开始运转，直到整个系统关闭才退出。

【例 11-3】为进程加上 daemon 属性：

```
import multiprocessing
import time

def worker(interval):
    print("work start:{0}".format(time.ctime()))
    time.sleep(interval)
    print("work end:{0}".format(time.ctime()))

if __name__ == "__main__":
    p = multiprocessing.Process(target = worker, args = (3,))
    p.daemon = True
    p.start()
    print ("end!")
```

输出结果：

```
end!
```

之所以出现这种结果，是因为主进程在运行完 print("end!")语句后就结束了，而当 daemon 属性设置为 True 时，主进程结束时会将其子进程连同主进程一同结束，而不论其子进程是否已经运行完毕。所以 worker 进程尚未运行完就被终止了。如果想设置 daemon 执行完结束，就需要引入 join()方法。

join([timeout])方法：阻塞当前进程，等待被调用的子进程结束再执行主进程的后续代码。其中 timeout 参数用来设置最长等待时间，以免子进程运行时间过长，影响其他代码的运行。

【例 11-4】在程序中加入 join 方法：

```
import multiprocessing
```

```
def worker 1():
   for i in range(5):
      print("worker 1:"+str(i))

def worker 2():
   for i in range(5):
      print("worker 2:"+str(i))

if   name   == "  main  ":
    p1 = multiprocessing.Process(target = worker 1, args = ())
    p2 = multiprocessing.Process(target = worker 2, args = ())
    p1.start()
    p1.join()
    p2.start()
```

输出结果：

```
worker_1:0
worker_1:1
worker_1:2
worker_1:3
worker_1:4
worker_2:0
worker_2:1
worker_2:2
worker_2:3
worker_2:4
```

通过上面的例子可以知道，如果没有加 join()语句，两个函数的输出结果应该是重叠输出的。但加入 join()语句后，它对后面的代码进行阻塞，只有当 worker_1 进程完全结束后才会执行后续代码。

所以只要在例 11-3 的 p.start()语句后加上 p.join()，结果就会变为：

```
work start:Wed Aug 23 15:52:45 2017
work end:Wed Aug 23 15:52:48 2017
end!
```

11.3.3　进程间通信技术 Queue 和 Pipe

1. Queue 通信

Queue 是安全的多进程队列，使用 Queue 可以实现多进程之间的数据传递，使用其中的 Queue.put()方法可以将数据插入到消息队列中。put()方法还有两个可选参数：blocked 和 timeout。blocked 默认值为 True，此时若 timeout 为正值，则该方法会阻塞 timeout 指定的时间，直到该队列有剩余的空间；如果超时，会抛出 Queue.Full 异常。如果 blocked 为

False，但该 Queue 已满，会立即抛出 Queue.Full 异常。

当消息保存到消息队列中时，其他进程可以通过 Queue.get()方法从消息队列中取出一个消息并删除。同样地，get()方法也有两个可选参数：blocked 和 timeout。如果 blocked 为 True(默认值)，并且 timeout 为正值，那么在等待时间内如果没有取到任何元素，就会抛出 Queue.empty 异常。如果 blocked 为 False，有两种情况存在，如果 Queue 有一个值可用，则立即返回该值；否则，如果队列为空，则立即抛出 Queue.empty 异常。

【例 11-5】使用 Queue 进行进程间通信：

```
import multiprocessing

def writer_proc(q):
    try:
        q.put(1, block = False)
    except:
        pass

def reader_proc(q):
    try:
        print (q.get(block = False))
    except:
        pass

if __name__ == "__main__":
    q = multiprocessing.Queue()
    writer = multiprocessing.Process(target=writer_proc, args=(q,))
    writer.start()

    reader = multiprocessing.Process(target=reader_proc, args=(q,))
    reader.start()

    reader.join()
    writer.join()
```

输出结果：

```
1
```

在上面的例子中，首先运行一个进程 writer_proc()，向 Queue 消息队列中写入一个值 1，消息通过 Queue 队列被进程 reader_proc()读取并打印出来。

2. Pipe 管道

Pipe 方法返回(conn1, conn2)，代表一个管道的两个端。Pipe 方法有 duplex 参数，如果 duplex 参数为 True(默认值)，那么这个管道是全双工模式，也就是说 conn1 和 conn2 均可收发。若 duplex 为 False，则 conn1 只负责接收消息，conn2 只负责发送消息。

send 和 recv 方法分别是发送和接收消息的方法。例如，在全双工模式下，可以调用 conn1.send 发送消息，调用 conn1.recv 接收消息。如果没有消息可接收，recv 方法会一直阻塞。如果管道已经被关闭，那么 recv 方法会抛出 EOFError 错误。

【例 11-6】使用 Pipe 管道通信：

```
import multiprocessing
import time

def proc1(pipe):
    while True:
        for i in range(10000):
            print ("send: %s" %(i))
            pipe.send(i)
            time.sleep(1)

def proc2(pipe):
    while True:
        print ("proc2 rev:", pipe.recv())
        time.sleep(1)

def proc3(pipe):
    while True:
        print ("proc3 rev:", pipe.recv())
        time.sleep(1)

if __name__ == "__main__":
    pipe = multiprocessing.Pipe()
    p1 = multiprocessing.Process(target=proc1, args=(pipe[0],))
    p2 = multiprocessing.Process(target=proc2, args=(pipe[1],))
    p3 = multiprocessing.Process(target=proc3, args=(pipe[1],))

    p1.start()
    p2.start()
    p3.start()
```

输出结果：

```
send: 0
proc3 rev: 0
send: 1
proc2 rev: 1
send: 2
proc3 rev: 2
send: 3
proc2 rev: 3
send: 4
proc3 rev: 4
send: 5
proc2 rev: 5
send: 6
proc3 rev: 6
send: 7
proc2 rev: 7
send: 8
proc3 rev: 8
send: 9
proc2 rev: 9
send: 10
```

在上面的例子中，定义了一个消息发送进程 proc1()，proc1()通过 Pipe 管道的一端每秒钟向消息管道中发送一个消息。进程 proc2()和 proc3()同时在 Pipe 管道的另一端交替接收消息。

11.3.4 使用进程池 pool

在利用 Python 进行系统管理的时候，特别是同时操作多个文件目录，或者远程控制多台主机时，并行操作可以节约大量的时间。当被操作对象数目不大时，可以直接利用 Multiprocessing 中的 Process 动态成生多个进程。十几个还好，但如果有上百个、上千个目标，手动地去限制进程数量却又太过繁锁，此时，可以发挥进程池的功效。

pool 可以提供指定数量的进程，供用户调用，当有新的请求提交到 pool 中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到规定的最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行它。

【例 11-7】设置进程池个数：

```
import multiprocessing
import time

def func(msg):
    print ("msg:", msg)
    time.sleep(3)
    print ("end")

if   name   == "  main  ":
    pool = multiprocessing.Pool(processes = 3)
    for i in range(4):
        msg = ("hello %d" %(i))
        pool.apply async(func, (msg, ))
          #维持执行的进程总数为 processes，当一个进程执行完毕后，会添加新的进程进去

    print ("Mark~ Mark~ Mark~")
    pool.close()
    pool.join()   #调用 join 之前，先调用 close 函数，否则会出错。
              #执行完 close 后，不会有新的进程加入到 pool，join 函数等待所有子进程结束
    print ("Sub-process(es) done.")
```

执行结果：

```
Mark~ Mark~ Mark~
msg: hello 0
msg: hello 1
msg: hello 2
end
msg: hello 3
end
end
end
Sub-process(es) done.
```

从上面的输出结果可以看到，只有当前三个进程中有进程执行完毕后，才会将新的进程加入进程池执行。

【例 11-8】使用进程池(阻塞)：

```
import multiprocessing
import time

def func(msg):
    print ("msg:", msg)
    time.sleep(3)
    print ("end")

if __name__ == "__main__":
    pool = multiprocessing.Pool(processes = 3)
    for i in range(4):
        msg = ("hello %d" %(i))
        pool.apply(func, (msg, ))
          #维持执行的进程总数为processes，当一个进程执行完毕后，会添加新的进程进去

    print ("Mark~ Mark~ Mark~")
    pool.close()
    pool.join()    #调用join之前，先调用close()函数，否则会出错。执行完close后，
                   #不会有新的进程加入到pool，join()函数等待所有子进程结束
    print ("Sub-process(es) done.")
```

执行结果：

```
msg: hello 0
end
msg: hello 1
end
msg: hello 2
end
msg: hello 3
end
Mark~ Mark~ Mark~
Sub-process(es) done.
```

函数解释：

- apply_async(func[, args[, kwds[, callback]]])是非阻塞的，apply(func[, args[, kwds]])是阻塞的(要理解区别，可以看例 11-7 和例 11-8 结果的区别)。
- close()关闭 pool，使其不再接受新的任务。
- terminate()结束工作进程，不再处理未完成的任务。
- join()使主进程阻塞，等待子进程的退出，join 方法要在 close 或者 terminate 之后使用。

执行说明：创建一个进程池 pool，并设定进程的数量为 3，xrange(4)会相继产生 4 个对象[0, 1, 2, 3]，这 4 个对象被提交到 pool 中，因 pool 指定进程数为 3，所以 0、1、2 会

直接送到进程池中执行，当其中一个执行完毕后才空出一个进程处理对象 3，所以输出“msg: hello 3”出现在“end”之后。因为为非阻塞，主函数会自己执行自己的，不理睬进程的执行，所以运行完 for 循环后直接输出“Mark~ Mark~ Mark~”，主程序在 pool.join() 处等待各个进程的结束。

【例 11-9】使用进程池，并关注结果：

```
import multiprocessing
import time

def func(msg):
    print ("msg:", msg)
    time.sleep(3)
    print ("end")
    return "done" + msg

if   name   == "  main  ":
    pool = multiprocessing.Pool(processes=4)
    result = []
    for i in range(3):
        msg = "hello %d" %(i)
        result.append(pool.apply async(func, (msg, )))
    pool.close()
    pool.join()
    for res in result:
        print (":::", res.get())
    print ("Sub-process(es) done.")
```

执行结果：

```
msg: hello 0
msg: hello 1
msg: hello 2
end
end
end
::: donehello 0
::: donehello 1
::: donehello 2
Sub-process(es) done.
```

从上面的运行结果可以看到，代码中使用了一个列表 result 收集各个进程返回的信息，并通过 for 循环输出收集到的信息。

【例 11-10】使用多个进程池：

```
import multiprocessing
import os, time, random

def Lee():
    print ("\nRun task Lee-%s" %(os.getpid())) #os.getpid()获取当前的进程的 ID
    start = time.time()
    time.sleep(random.random() * 10) #random.random()随机生成 0~1 之间的小数
```

```
    end = time.time()
    print ('Task Lee, runs %0.2f seconds.' %(end - start))

def Marlon():
    print ("\nRun task Marlon-%s" %(os.getpid()))
    start = time.time()
    time.sleep(random.random() * 40)
    end=time.time()
    print ('Task Marlon runs %0.2f seconds.' %(end - start))

def Allen():
    print ("\nRun task Allen-%s" %(os.getpid()))
    start = time.time()
    time.sleep(random.random() * 30)
    end = time.time()
    print ('Task Allen runs %0.2f seconds.' %(end - start))

def Frank():
    print ("\nRun task Frank-%s" %(os.getpid()))
    start = time.time()
    time.sleep(random.random() * 20)
    end = time.time()
    print ('Task Frank runs %0.2f seconds.' %(end - start))

if   name  ==' main  ':
    function list= [Lee, Marlon, Allen, Frank]
    print ("parent process %s" %(os.getpid()))

    pool=multiprocessing.Pool(4)
    for func in function list:
        pool.apply async(func)     #Pool 执行函数，apply 执行函数，
                                   #当有一个进程执行完毕后，会添加一个新的进程到 pool 中

    print ('Waiting for all subprocesses done...')
    pool.close()
    pool.join()    #调用 join 之前，一定要先调用 close()函数，否则会出错，
                #close()执行后不会有新的进程加入到 pool，join 函数等待所有子进程结束
    print ('All subprocesses done.')
```

执行结果：

```
parent process 4936
Waiting for all subprocesses done...

Run task Lee-2488

Run task Marlon-6020

Run task Allen-9912

Run task Frank-10112
Task Lee, runs 6.84 seconds.
Task Frank runs 13.61 seconds.
Task Allen runs 15.01 seconds.
Task Marlon runs 35.39 seconds.
All subprocesses done.
```

在上面的例子中，我们通过 for 循环同时运行了 4 个进程池，每个进程池运行一个进程。从输出结果可以看出，各个进程间为并行运行。

11.4　多线程编程

Python 通过两个标准库 thread 和 threading 提供对线程的支持。thread 提供了低级别的、原始的线程以及一个简单的锁。threading 通过对 thread 模块进行二次封装，提供了更方便的 API 来操作线程。这里将就 threading 模块展开详细介绍。threading 模块的常用方法如表 11-2 所示。

表 11-2　threading 模块的常用方法

方法名称	说　明
threading.Thread	threading 模块中最重要的类之一，可以使用它来创建线程
threading.RLock	在 threading 模块中定义的锁，允许在同一线程中被多次 acquire
threading.Lock	在 threading 模块中定义的锁，不允许在同一线程中被多次 acquire
threading.Condition	提供了比 Lock、RLock 更高级的功能，能够控制复杂的线程同步问题
threading.Time	可以在指定时间间隔后执行某个操作
threading.currentThread()	获取当前的线程对象
threading.enumerate()	获取当前所有活动线程的列表
threading.settrace(func)	设置一个跟踪函数，用于在 run()执行之前被调用

11.4.1　Thread 对象

Thread 是 threading 模块中最重要的类之一，可以使用它来创建线程。

有两种方式来创建线程：一种是通过继承 Thread 类，重写它的 run()方法；另一种是创建一个 threading.Thread 对象，在它的初始化函数(__init__)中将可调用对象作为参数传入。下面分别举例说明。先来看看通过继承 threading.Thread 类来创建线程的例子。

【例 11-11】通过继承 threading.Thread 类来创建线程：

```
import threading, time

count = 0

class Counter(threading.Thread):
    def __init__(self, lock, threadName):
        '''@summary: 初始化对象。
           @param lock: 锁对象。
           @param threadName: 线程名称
        '''
        super(Counter, self).__init__(name = threadName)
          #注意：一定要显式地调用父类的初始化函数
```

```
        self.lock = lock

    def run(self):
        '''@summary: 重写父类 run()方法，在线程启动后执行该方法内的代码
        '''
        global count
        self.lock.acquire()
        for i in range(10000):
            count = count + 1
        self.lock.release()

lock = threading.Lock()

for i in range(5):
    Counter(lock, "thread-" + str(i)).start()

time.sleep(2)  #确保线程都执行完毕
print (count)
```

执行结果：

```
50000
```

在代码中，创建了一个 Counter 类，它继承了 threading.Thread。初始化函数接收两个参数，一个是锁对象，另一个是线程的名称。在 Counter 中，重写了从父类继承的 run 方法，run 方法将一个全局变量逐一地增加到 10000。在接下来的代码中，创建了 5 个 Counter 对象，分别调用其 start 方法，最后打印结果。

这里要说明一下 run 方法和 start 方法，它们都是从 Thread 继承而来的，run 方法将在线程开启后执行，可以把相关的逻辑写到 run 方法中(通常把 run 方法称为活动(Activity))；start 方法用于启动线程。

再看看另外一种创建线程的方法。

【例 11-12】另一种创建线程的方法：

```
import threading, time

count = 0
lock = threading.Lock()

def doAdd():
    '''@summary: 将全局变量 count 逐一增加 10000
    '''
    global count, lock
    lock.acquire()
    for i in range(10000):
        count = count + 1
    lock.release()

for i in range(5):
    threading.Thread(target = doAdd, args = (), name = 'thread-' +
str(i)).start()
```

```
time.sleep(2)  #确保线程都执行完毕
print (count)
```

执行结果：

```
50000
```

在这段代码中，定义了方法 doAdd，它将全局变量 count 逐一地增加到 10000。然后创建了 5 个 Thread 对象，把函数对象 doAdd 作为参数传给它的初始化函数，再调用 Thread 对象的 start 方法，线程启动后，将执行 doAdd 函数。

这里有必要介绍一下 threading.Thread 类的初始化函数原型：

```
def  init  (self, group=None, target=None, name=None, args=(),
kwargs={})
```

参数 group 是预留的，用于将来扩展。

参数 target 是一个可调用对象，在线程启动后执行。

参数 name 是线程的名字，默认值为 Thread-N，N 是一个数字。

参数 args 和 kwargs 分别表示调用 target 时的参数列表和关键字参数。

Thread 类还定义了表 11-3 所示的常用方法与属性。

表 11-3　Thread 类的常用方法与属性

常用方法与属性	说　明
Thread.getName() Thread.setName()	用于获取和设置线程的名称
Thread.ident	获取线程的标识符。线程标识符是一个非零整数，只有在调用了 start()方法之后，该属性才有效，否则它只返回 None
Thread.is_alive() Thread.isAlive()	判断线程是否是激活的(alive)。从调用 start()方法启动线程，到 run()方法执行完毕或遇到未处理异常而中断这段时间内，线程是激活的
Thread.join([timeout])	调用 Thread.join 将会使主调线程阻塞，直到被调用线程运行结束或超时。参数 timeout 是一个数值类型，表示超时时间，如果未提供该参数，那么主调线程将一直阻塞到被调线程结束

11.4.2　thread 锁

在 threading 模块中定义了两种类型的锁：threading.Lock 和 threading.RLock，它们之间有一点细微的区别，我们通过比较下面的两段代码来说明。

【例 11-13】两种锁方法的对比：

```
import threading
lock = threading.Lock()    #Lock 对象
lock.acquire()
lock.acquire()  #产生了死锁
lock.release()
```

```
lock.release()

import threading
rLock = threading.RLock()  #RLock 对象
rLock.acquire()
rLock.acquire()    #在同一线程内，程序不会阻塞
rLock.release()
rLock.release()
```

这两种锁的主要区别是：RLock 允许在同一线程中被多次 acquire，而 Lock 却不允许出现这种情况。

注意： 如果使用 RLock，那么 acquire 和 release 必须成对出现，即调用了 n 次 acquire，必须调用 n 次 release 才能真正释放所占用的锁。

除了上面的两种锁，threading 模块还提供了另一种更加实用的锁 Condition。可以把 Condition 理解为一把高级的锁，它提供了比 Lock 和 RLock 更高级的功能，允许控制复杂的线程同步问题。threading.Condition 在内部维护一个锁对象(默认是 RLock)，可以在创建 Condition 对象的时候把锁对象作为参数传入。Condition 也提供了 acquire 和 release 方法，其含义与锁的 acquire 和 release 方法一致，其实它只是简单地调用内部锁对象所对应的方法而已。

Condition 还提供了如下方法(特别要注意，这些方法只有在占用锁(acquire)之后才能调用，否则将会报 RuntimeError 异常)。

(1) Condition.wait([timeout])。

wait 方法释放内部所占用的锁，同时线程被挂起，直至接收到通知被唤醒或超时(如果提供了 timeout 参数的话)。当线程被唤醒并重新占有锁的时候，程序才会继续执行下去。

(2) Condition.notify()。

唤醒一个挂起的线程(如果存在挂起的线程)。注意 notify()方法不会释放所占用的锁。

(3) Condition.notify_all()。

唤醒所有挂起的线程(如果存在挂起的线程)。注意这些方法不会释放所占用的锁。

11.5　案例实训：捉迷藏游戏设计

现在通过一个捉迷藏游戏程序来具体介绍 threading 的基本使用。假设这个游戏由两个人来玩，一个藏(Hider)，一个找(Seeker)。游戏的规则如下。

(1) 游戏开始之后，Seeker 先把自己眼睛蒙上，蒙上眼睛后，就通知 Hider。

(2) Hider 接收通知后，开始找地方将自己藏起来，藏好后再通知 Seeker 找。

(3) Seeker 接收到通知后，就开始找 Hider。

Hider 和 Seeker 都是独立的个体，在程序中用两个独立的线程来表示，在游戏过程中，两者之间的行为有一定的时序关系，通过 Condition 来控制这种时序关系：

```
import threading, time
class Seeker(threading.Thread):
    def __init__(self, cond, name):
```

```
        super(Seeker, self).__init__()
        self.cond = cond
        self.name = name

    def run(self):
        time.sleep(1) #确保先运行 Hider 中的方法

        self.cond.acquire() #b
        print (self.name + ': 我已经把眼睛蒙上了')
        self.cond.notify()
        self.cond.wait() #c
                        #f
        print (self.name + ': 我找到你了 ~ ~')
        self.cond.notify()
        self.cond.release()
                            #g
        print (self.name + ': 我赢了')  #h

class Hider(threading.Thread):
    def __init__(self, cond, name):
        super(Hider, self).__init__()
        self.cond = cond
        self.name = name
    def run(self):
        self.cond.acquire()
        self.cond.wait()    #a    #释放对锁的占用，同时线程挂起在这里，直到被
notify 并重新占有锁
                            #d
        print (self.name + ': 我已经藏好了，你快来找我吧')
        self.cond.notify()
        self.cond.wait()    #e
                            #h
        self.cond.release()
        print (self.name + ': 被你找到了，哎~')

cond = threading.Condition()
seeker = Seeker(cond, 'seeker')
hider = Hider(cond, 'hider')
seeker.start()
hider.start()
```

运行结果：

```
seeker: 我已经把眼睛蒙上了
hider: 我已经藏好了，你快来找我吧
seeker: 我找到你了 ~_~
seeker: 我赢了
hider: 被你找到了，哎~
```

本 章 小 结

在本章中，我们主要对多进程和多线程编程及其相关技术进行了介绍，并通过几个实际的应用来帮助读者深入了解多进程和多线程的使用方法。在程序中合理地利用这两种技术，不但可以大幅地提升程序的运算效率，而且可以使程序实现变得更加轻松。

习　　题

1. 填空题

(1) (　　　)是资源分配的最小单位，(　　　)是 CPU 调度的最小单位。

(2) 运行中的进程可能具有以下三种基本状态：(　　　)、(　　　)、(　　　)。

(3) Multiprocessing 提供了两种进程间通信的组件，分别是(　　　)和(　　　)，其中(　　　)通信方式支持全双工模式。

(4) thread 锁有三种方法，分别为(　　　)、(　　　)、(　　　)。

2. 选择题

(1) 与多进程相比，多线程在(　　)方面较为占优。

A. 可靠性　　　　B. 数据共享、同步

C. 创建、销毁、切换　　　　D. 编程、调试

(2) 阅读下面的程序，程序的输出结果可能为(　　)。

```
import multiprocessing

def worker 1():
    print ("worker 1")
def worker 2():
    print ("worker 2")
def worker 3():
    print ("worker 3")

if   name    == "  main  ":
    p1 = multiprocessing.Process(target = worker 1, args = ())
    p2 = multiprocessing.Process(target = worker 2, args = ())
    p3 = multiprocessing.Process(target = worker 3, args = ())
    p1.start()
    p2.start()
    p3.start()
```

A.	B.	C.	D.
worker_1 worker_2 worker_3	worker_2 worker_1 worker_3	worker_3 worker_2 worker_1	都有可能

3. 问答题

(1) 什么是线程？

(2) 什么是进程？



第 12 章

网络编程

本章要点

(1) 计算机网络基础知识。
(2) socket 通信技术。
(3) urllib 库及其使用。
(4) 端口扫描器编程。
(5) 简单网络爬虫编程。

学习目标

(1) 理解基本的计算机网络知识。
(2) 掌握 socket 通信技术及其应用方法。
(3) 掌握 urllib 库及其应用方法。
(4) 通过端口扫描器和简单的网络爬虫例子综合应用所学内容。

Python 语言的快速编程特性，使我们可以快速地编写一些小型工具。本章主要介绍 Python 语言在计算机网络通信方面的应用，首先介绍一些计算机网络的相关知识，然后通过一个端口扫描器程序介绍如何利用 socket 进行计算机之间的通信，同时简要介绍 Python 提供的用于网络通信的大量的库，最后通过一个简单的网络爬虫程序介绍 Python 在网络数据收集方面的应用。

12.1 计算机网络基础知识

为了更好地学习和掌握后续内容，本章一开始首先简要介绍一下与计算机网络相关的基础知识。如果想对其做更深入的了解，请参考有关计算机网络的文献，例如《计算机网络(第 5 版)》、《计算机网络：自顶向下方法(第 6 版)》、《深入理解计算机网络》、《TCP/IP 详解》等(参考文献[22~25])。

1. 计算机网络体系结构

计算机网络体系结构可以定义为网络协议的层次划分与各层协议的集合，同一层中的协议根据该层所要实现的功能来确定。各对等层之间的协议功能由相应的底层提供服务完成。目前主要的计算机网络体系结构是 ISO/OSI 网络体系结构和 TCP/IP 协议簇。两种体系结构都采用了分层设计和实现的方式，在这里我们主要介绍 TCP/IP 协议簇。

传输控制协议/网间协议(Transmission Control Protocol/Internet Protocol，TCP/IP)定义了主机如何连入因特网及数据如何在它们之间传输的标准。从字面意思来看，TCP/IP 是 TCP 和 IP 协议的合称，但实际上，TCP/IP 协议是指因特网整个 TCP/IP 协议簇。不同于 OSI 模型的七个分层，TCP/IP 协议参考模型把所有的 TCP/IP 系列协议归类到四个抽象层中：应用层、传输层、网络层和链路层，通信过程如图 12-1 所示。

现对各层协议的功能做如下说明。

(1) 链路层。

链路层对应 OSI 模型的数据链路层和物理层，负责向网络媒体发送和接收 TCP/IP 数

据包。

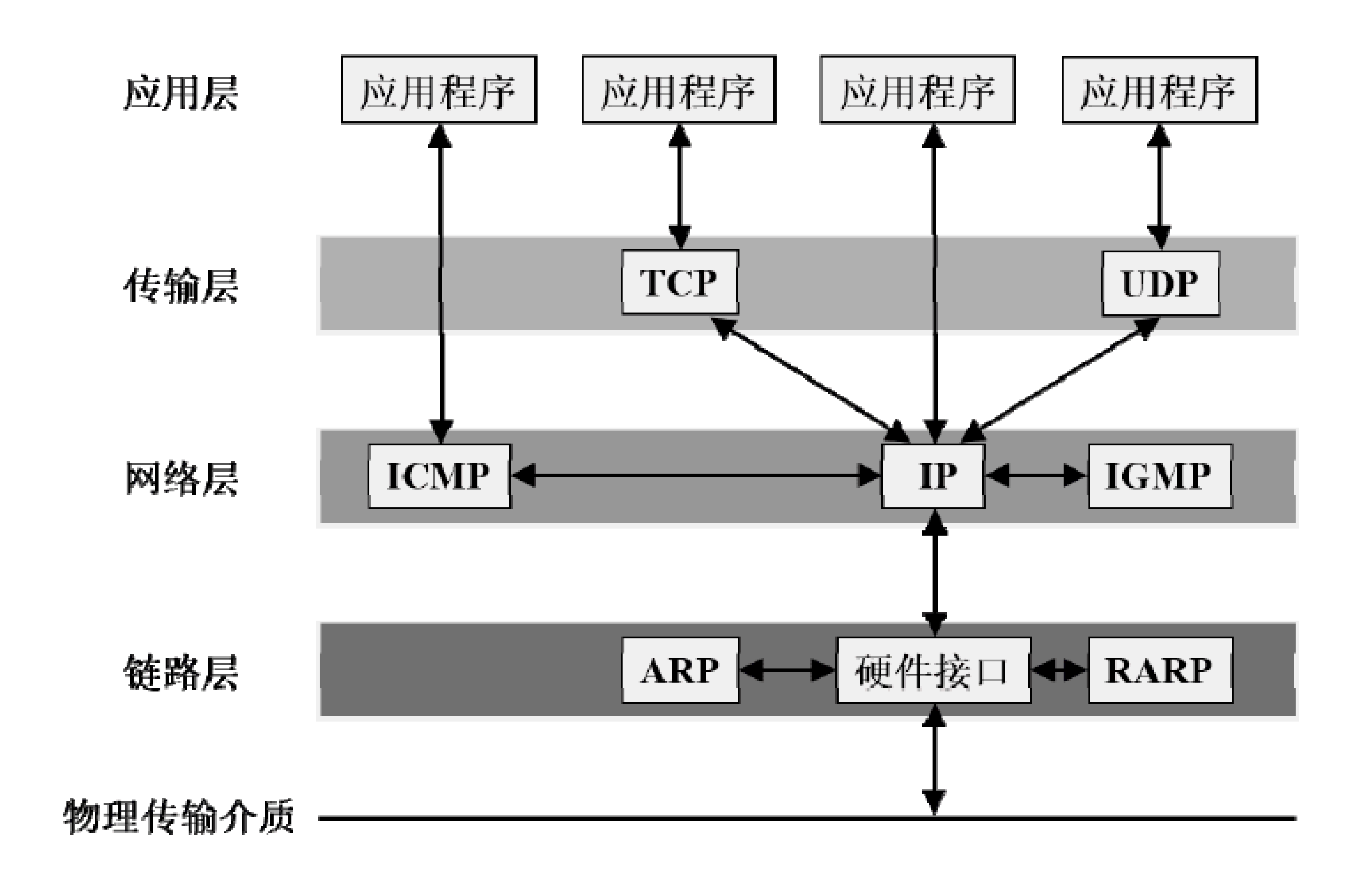


图 12-1　TCP/IP 协议通信过程

(2)　网络层。

网络层是整个体系结构的关键部分，其功能是使主机可以把分组发往任何网络，并使分组独立地传向目的地。这些分组可能经由不同的网络路径，到达的顺序和发送的顺序也可能不同。该层功能与 OSI 参考模型的网络层功能相似。

(3)　传输层。

传输层使源端和目的端机器上的对等实体进行会话。在这一层定义了两个端到端的协议：传输控制协议(TCP)和用户数据包协议(UDP)。TCP 是面向连接的协议，它提供可靠的报文传输和对上层应用的连接服务。为此，除了基本的数据传输外，它还有可靠性保证、流量控制、多路复用、优先权和安全性控制等功能。UDP 是面向无连接的不可靠传输协议，主要用于不需要 TCP 的排序和流量控制等功能的应用程序。

(4)　应用层。

应用层是 TCP/IP 协议的最高层，对应于 OSI 模型的应用层、表示层和会话层。应用层包含所有的高层协议，包括虚拟终端协议(Telnet)、文件传输协议(FTP)、简单邮件传输协议(SMTP)、域名服务(DNS)、网上新闻传输协议(NNTP)和超文本传输协议(HTTP)等。Telnet 允许一台机器上的用户登录到远程机器上，并进行工作；FTP 提供有效地将文件从一台机器移到另一台机器上的方法；SMTP 用于电子邮件的收发；DNS 用于把主机名映射到网络地址；NNTP 用于新闻的发布、检索和获取；HTTP 用于在 WWW 上获取主页。

2. 网络协议

网络协议是计算机网络中为进行数据交换而建立的规则、标准或约定的集合。网络协议的三要素为语法、语义和时序。

简单地讲，语义表示要做什么，语法表示要怎么做，时序规定了各种事件出现的顺序。表 12-1 列示了计算机网络中常用的基本概念。

表 12-1　计算机网络中常用的基本概念

基本概念	解　释
语义	语义是解释控制信息每个部分的意义。它规定了需要发出何种控制信息，以及完成的动作与做出什么样的响应
语法	语法是用户数据与控制信息的结构与格式，及数据出现的顺序
时序	时序是对事件发生顺序的详细说明(也可称为“同步”)
域名系统 (Domain Name System，DNS)	DNS 是由解析器和域名服务器组成的。域名服务器是指保存有该网络中所有主机的域名和对应 IP 地址，并具有将域名转换为 IP 地址功能的服务器
文件传输协议 (File Transfer Protocol，FTP)	FTP 是 TCP/IP 协议簇中的协议之一，用于在 Internet 上控制文件的双向传输。同时，它也是一个应用程序。用户可以通过它把自己的 PC 与世界各地所有运行 FTP 协议的服务器相连，访问服务器上的大量程序和信息
超文本传输协议(HyperText Transfer Protocol，HTTP)	HTTP 是客户端浏览器或其他程序与 Web 服务器之间的应用层通信协议。在 Internet 上的 Web 服务器上存放的都是超文本信息，客户机需要通过 HTTP 协议传输所要访问的超文本信息。HTTP 包含命令和传输信息，不仅可用于 Web 访问，也可以用于其他因特网/内联网应用系统之间的通信，从而实现各类应用资源超媒体访问的集成
简单邮件传输协议(Simple Mail Transfer Protocol，SMTP)	SMTP 是一种 TCP 协议支持的提供可靠且有效的电子邮件传输的应用层协议。SMTP 是建立在 TCP 上的一种邮件服务，主要用于传输系统之间的邮件信息并提供与来信有关的通知
地址解析协议(Address Resolution Protocol，ARP)	实现 IP 地址与 MAC 地址间的转换
简单文件传输协议(Trivial File Transfer Protocol，TFTP)	TFTP 是 TCP/IP 协议簇中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务
简单网络管理协议 (Simple Network Management Protocol，SNMP)	该协议能够支持网络管理系统，用以监测连接到网络上的设备是否有任何引起管理上关注的情况
Telnet	Telnet 协议是 TCP/IP 协议簇中的一员，是 Internet 远程登录服务的标准协议和主要方式
用户数据包协议 (User Datagram Protocol，UDP)	UDP 是 OSI 参考模型中一种无连接的传输层协议，它主要用于不要求分组顺序到达的传输中，分组传输顺序的检查与排序由应用层完成，提供面向事务的简单不可靠信息传送服务
传输控制协议(Transmission Control Protocol，TCP)	TCP 是一种面向连接的、可靠的、基于字节流的传输层通信协议
IP 地址	IP 地址是 IP 协议提供的一种统一的地址格式，它为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。如果把个人电脑比作一台电话，那么 IP 地址就相当于电话号码

12.2 socket 通信技术

12.2.1 什么是 socket

socket 又称“套接字”，应用程序通常通过“套接字”向网络发出请求或者应答网络请求，实现主机间或者一台计算机上的进程间通信。

socket 非常类似于电话插座。

以一个国家级电话网为例，电话的通话双方相当于相互通信的两个进程，区号是它的网络地址；区内一个单位的交换机相当于一台主机，主机分配给每个用户的局内号码相当于 socket 号。

任何用户在通话之前，首先要占有一部电话机，相当于申请一个 socket。同时要知道对方的号码，相当于对方有一个固定的 socket。然后向对方拨号呼叫，相当于发出连接请求(假如对方不在同一区内，还要拨对方区号，相当于给出网络地址)。假如对方在场并空闲(相当于通信的另一主机开机且可以接受连接请求)，拿起电话话筒，双方就可以正式通话，相当于连接成功。双方通话的过程，是一方向电话机发出信号和对方从电话机接收信号的过程(相当于向 socket 发送数据和从 socket 接收数据)。通话结束后，一方挂断电话机(相当于关闭 socket，撤消连接)。

12.2.2 连接过程

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认。

(1) 服务器监听：是指服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监听网络状态。

(2) 客户端请求：是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

(3) 连接确认：是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

12.2.3 socket 模块

Python 提供了两个基本的 socket 模块，一个是 socket，它提供了标准的 BSD socket API；另一个是 socketServer，它提供了服务器中心类，可以简化网络服务器的开发。socket 模块的部分类方法如表 12-2 所示。

表 12-2　socket 模块的部分类方法

类 方 法	说　明
socket.socket(family, type[,proto])	创建并返回一个新的 socket 对象
socket.getfqdn(name)	将使用点号分隔的 IP 地址字符串转换成一个完整的域名
socket.gethostbyname(hostname)	将主机名解析为一个使用点号分隔的 IP 地址字符串
socket.gethostbyname_ex(name)	它返回一个包含三个元素的元组，从左到右分别是给定地址的主机名、同一 IP 地址的可选的主机名的一个列表、关于同一主机的同一接口的其他 IP 地址的一个列表(列表可能都是空的)
socket.gethostbyaddr(address)	作用与 gethostbyname_ex 相同，只是提供给它的参数是一个 IP 地址字符串
socket.getservbyname(service, protocol)	它要求一个服务名(如'telnet'或'ftp')和一个协议(如'tcp'或'udp')，返回服务所使用的端口号
socket.fromfd(fd, family, type)	从现有的文件描述符创建一个 socket 对象

12.2.4　socket 函数

socket 函数是一种可用于根据指定的地址族、数据类型和协议来分配一个套接字及其所用资源的函数，表 12-3 为常用的 socket 函数。

表 12-3　常用的 socket 函数

socket 函数	说　明
服务端 socket 函数	
s.bind(address)	将套接字绑定到地址，在 AF_INET 下，以元组(host, port)的形式表示地址
s.listen(backlog)	开始监听 TCP 传入连接。backlog 指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为 1，大部分应用程序设为 5 就可以了
s.accept()	接受 TCP 连接并返回(conn, address)，其中 conn 是新的套接字对象，可以用来接收和发送数据。address 是连接客户端的地址
客户端 Socket 函数	
s.connect(address)	连接到 address 处的套接字。一般 address 的格式为元组 (hostname, port)，如果连接出错，则返回 socket.error 错误
s.connect_ex(address)	功能与 connect(address)相同，但是成功返回 0，失败返回 errno 的值

续表

socket 函数	说　明
公共 socket 函数	
s.recv(bufsize[,flag])	接受 TCP 套接字的数据。数据以字符串形式返回，bufsize 指定要接收的最大数据量。flag 提供有关消息的其他信息，通常可以忽略
s.send(string[,flag])	发送 TCP 数据。将 string 中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于 string 的字节大小
s.sendall(string[,flag])	完整发送 TCP 数据。将 string 中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回 None，失败则抛出异常
s.recvfrom(bufsize[.flag])	接受 UDP 套接字的数据。与 recv()类似，但返回值是(data, address)。其中 data 是包含接收数据的字符串，address 是发送数据的套接字地址
s.sendto(string[,flag],address)	发送 UDP 数据。将数据发送到套接字，address 是形式为(ipaddr, port)的元组，指定远程地址。返回值是发送的字节数
s.close()	关闭套接字
s.getpeername()	返回连接套接字的远程地址。返回值通常是元组(ipaddr, port)
s.getsockname()	返回套接字自己的地址。通常是一个元组(ipaddr, port)
s.setsockopt(level,optname,value)	设置给定套接字选项的值
s.getsockopt(level,optname[.buflen])	返回套接字选项的值
s.settimeout(timeout)	设置套接字操作的超时期，timeout 是一个浮点数，单位是秒。值为 None 表示没有超时期
s.gettimeout()	返回当前超时期的值，单位是秒，如果没有设置超时期，则返回 None
s.fileno()	返回套接字的文件描述符
s.setblocking(flag)	如果 flag 为 0，则将套接字设为非阻塞模式，否则将套接字设为阻塞模式(默认值)。非阻塞模式下，如果调用 recv()没有发现任何数据，或 send()调用无法立即发送数据，那么将引起 socket.error 异常
s.makefile()	创建一个与该套接字相关联的文件

注意：

- TCP 协议发送数据时，已建立好 TCP 连接，所以不需要指定地址。UDP 是无连接的协议，每次发送都要指定是发给谁。
- 服务器端与客户端不能直接发送列表、元组、字典，需要使用 repr(data)将其字符串化。

12.2.5 socket 编程思路

1. 服务器端编程

(1) 创建 socket 对象，需要调用 socket 构造函数：

```
socket = socket.socket(family,type)
```

其中，family 参数代表地址族，可以为 AF_INET 或 AF_UNIX。AF_INET 族包括 Internet 地址，AF_UNIX 族用于同一台机器上的进程间通信。Type 参数代表套接字类型，可为 SOCK_STREAM(流套接字)和 SOCK_DGRAM(数据包套接字)。

(2) 将 socket 绑定到指定地址。这是通过 socket 对象的 bind 方法来实现的：

```
socket.bind(address)
```

由 AF_INET 创建的套接字，其地址 address 必须是一个双元素元组，格式是(host, port)，host 代表主机，port 代表端口号。如果端口号正在使用、主机名不正确或端口已被保留，bind 方法将引发 socket.error 异常。

(3) 使用 socket 套接字的 listen 方法接收连接请求：

```
socket.listen(backlog)
```

backlog 指定最多允许多少个客户连接到服务器。它的值至少为 1。收到连接请求后，这些请求需要排队，如果队列满，就拒绝请求。

(4) 服务器套接字通过 socket 的 accept 方法等待客户请求一个连接：

```
connection.address = socket.accept()
```

调用 accept 方法时，socket 会进入 waiting 状态。客户请求连接时，该方法建立连接并返回服务器。accept 方法返回一个含有两个元素的元组(connection, address)。第一个元素 connection 是新的 socket 对象，服务器必须通过它与客户通信；第二个元素 address 是客户的 Internet 地址。

(5) 处理阶段，服务器和客户端通过 send 和 recv 方法通信(传输数据)。服务器调用 send，并采用字节串格式(bytes)向客户端发送信息。send 方法返回已发送的字符个数。服务器使用 recv 方法从客户端接收信息。调用 recv 时，服务器必须指定一个整数，它对应于可通过本次方法调用来接收的最大数据量。recv 方法在接收数据时会进入 blocked 状态，最后返回一个字符串，用它表示收到的数据。如果发送的数据量超过了 recv 所允许的，数据会被截断。多余的数据将缓冲于接收端，以后调用 recv 时，多余的数据会从缓冲区删除(也包括自上次调用 recv 以来，客户端可能发送的其他任何数据)。

(6) 在传输结束后，服务器调用 socket 的 close 方法关闭连接。

2. 客户端编程

(1) 创建一个 socket 以连接服务器：

```
socket = socket.socket(family, type)
```

(2) 使用 socket 的 connect()方法连接服务器。对于 AF_INET 族，连接格式如下：

```
socket.connect(host,port)
```

其中，host 代表服务器主机名或 IP，port 代表服务器进程所绑定的端口号。如连接成功，客户端就可通过套接字与服务器通信；如果连接失败，会引发 socket.error 异常。

(3) 处理阶段，客户和服务器将通过 send 方法和 recv 方法通信。

(4) 传输结束，客户通过调用 socket 的 close 方法关闭连接。

3. 服务器端编程和客户端编程的例子

在下面的例子中，我们将编写一个简单的利用 socket 通信的程序。

【例 12-1】socket 服务端编程代码(server.py)：

```
import socket  #引入 socket 模块
from time import ctime  #引入时间模块

bufsize = 1024  #设置套接字大小
host = '127.0.0.1'  #设置 IP 地址
port = 8100  #设置端口号
address = (host, port)

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  #定义 socket 类型
server_sock.bind(address)  #绑定 ip 地址和端口号
server_sock.listen(1)  #开始监听

while True:
    print('waiting for connection...')
    clientsock,addr = server_sock.accept()
      #接收 TCP 连接请求，并返回客户端的 IP 地址
    print('received from :', addr)  #输出客户端的地址
    while True:
        data = str(clientsock.recv(bufsize), encoding="utf-8")
          #接收客户端发送的信息，并转换为字符串格式
        print(' received-->%s\n%s' %(ctime(), data))  #输出收到的信息
        data = input("send-->")  #输入发送的信息
        clientsock.send(bytes(data, encoding="utf-8")) #将信息发送给客户端
    clientsock.close()  #关闭连接
server_sock.close()  #关闭服务器
```

【例 12-2】socket 客户端编程代码(client.py)：

```
from socket import *
from time import ctime

bufsize = 1024  #设置套接字大小
host = '127.0.0.1'  #设置 IP 地址
port = 8100  #设置端口号
address = (host, port)
```

```
client_sock = socket(AF_INET, SOCK_STREAM)  #定义 socket 类型
client_sock.connect(address)  #连接指定的 IP 端口

while True:
    data = input("发送-->")  #输入要发送的信息
    if not data:  #如果要发送的信息为空，则终止链接
        break
    else:
        client_sock.send(bytes(data, encoding="utf-8"))
           #将要发送的信息转换为 bytes 格式，并发送
        data = str(client_sock.recv(bufsize), encoding="utf-8")
           #接收服务器返回的信息，并转化为字符格式
        print ('收到-->%s\n%s' %(ctime(), data))  #输出收到的信息
client_sock.close()  #关闭连接
```

12.3　编写一个端口扫描器

前面我们通过一个可以互发消息的 socket 程序初步了解了 socket 是如何访问一台计算机的。然而，在实际应用中，假如我们试图使用某个端口连接远程的主机，有时会收到一个“connection is refused”(拒绝连接)消息。大多数情况下，收到这个信息是因为远程主机中的端口服务停止运行了。在遇到这种情况时，可以利用 socket 通信技术查看端口是处于开启状态还是处于监听状态。

当然，更重要的是可以用来测试端口的可访问性，因为这为我们提高目标主机的安全性提供了保障。

大多数的网络攻击都起步于侦察——攻击者首先通过扫描目标服务器的端口号，查看是否有端口处于活跃状态，然后再根据端口制定攻击计划。例如，最近爆发的“比特币病毒”WannaCry，就是利用 Windows 操作系统 445 端口存在的漏洞进行传播的，它进行自我复制，并主动传播，是一种典型的“勒索”类病毒。被该病毒入侵后，用户主机系统内的图片、文档、音频、视频等几乎所有类型的文件都将被加密，并在桌面弹出勒索对话框，要求受害者支付价值数百美元的比特币到攻击者的比特币钱包。

所以，为了避免服务器或者主机被黑客攻击，关闭可能被攻击的端口是提高服务器安全性的必要手段。

下面是一个端口扫描器的例子，用于检测计算机的哪些端口处于开启状态。

【例 12-3】远程主机端口扫描器：

```
import argparse
import socket
import sys

def scan_ports(host, start_port, end_port):
    try:
        sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    except socket.error as err_msg:
        print("Socket creation failed.Error code : "
```

```
          + str(err msg[0]) + "Error message:" + err msg[1])
        sys.exit()
    try:
        remote ip = socket.gethostbyname(host)
    except socket.error as err msg:
        print(err msg)
        sys.exit()

    end port += 1
    for port in range(start port,end port):
        try:
            sock.connect((remote ip,port))
            print("Port " + str(port) + " is open")
            sock.close()
            sock = socket.socket(socket.AF INET, socket.SOCK STREAM)
        except socket.error:
            pass

if   name   == '  main  ':
    parser = argparse.ArgumentParser(description="Remote Port Scanner ")
    parser.add argument('--host', action="store", dest="host",
      default='localhost')
    parser.add argument('--start-port', action="store", dest="start port",
      default=1, type=int)
parser.add argument('--end-port', action="store", dest="end port",
default=100, type=int)

    given args = parser.parse args()
    host, start port, end port = given args.host, given args.start port,
given args.end port
    scan_ports(host, start_port, end_port)
```

在上面的例子中，首先建立一个 scan_ports()函数，该函数接收三个参数：主机名，起始端口和终止端口。

然后建立 socket 套接字，套接字建立成功后，使用 gethostbyname()函数找出主机的 IP 地址。

最后通过一个 for 循环扫描对应 IP 地址主机的 1~100 端口号。如果连接成功，则输出端口号，输出的结果可能如下：

```
Port 80 is open
>>>
```

一般情况下，Web 服务器可能位于 TCP 80 端口，电子邮件服务可能在 TCP 25 端口，FTP 服务器可能在 TCP 21 端口。在扫描出所有开放的端口后，就可以根据实际情况关闭不需要的端口以提高服务器的安全性了。

12.4　简单网络爬虫的实现

Python 另一个强大的方面就是它的网络访问和网络数据收集功能。利用 Python 自带的

urllib 库，我们可以轻松地访问网络数据并将其保存下来。

12.4.1 什么是网络爬虫

网络爬虫(Web Crawler)是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本。网络爬虫又称为网页蜘蛛、网络机器人，在 FOAF 社区中，人们更经常称其为网页追者。另外，它还有一些不常使用的名字，如网络蚂蚁、自动索引、模拟程序或者蠕虫等。

网络爬虫广泛地应用于搜索引擎领域和数据收集方面。像 Google 公司和百度公司这种大型的搜索引擎公司更是拥有自己的大型分布式高效率网络爬虫。开源的网络爬虫框架有 Larbin、Heritrix、Scrapy、Crawler4j 等，感兴趣的读者可以参阅相关的文献，因篇幅所限，本书不再一一介绍。本节仅介绍 Python 网络爬虫的基础模块 urllib 库。

12.4.2 浏览网页的过程

在介绍 urllib 库之前，需要先了解一下浏览网络的过程。

使用浏览器浏览网页时，只要输入网页地址或者单击网络链接，就会弹出一个网站的页面。

例如浏览百度图片的网站时，会看到几张图片和百度图片的搜索框，如图 12-2 所示。

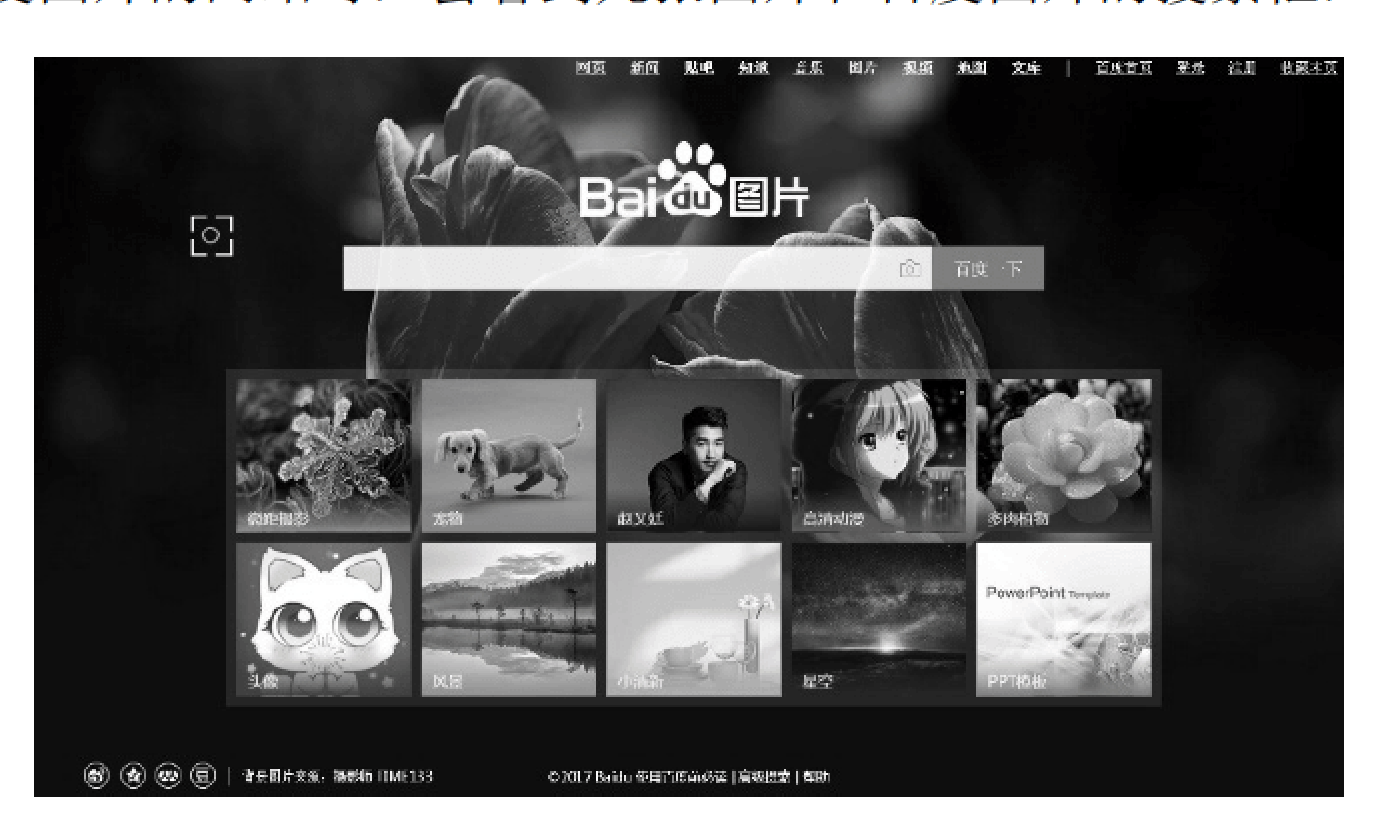


图 12-2　浏览器解析后的效果

其实，这个过程是用户在输入完网址后，浏览器将网址发送给 DNS 服务器，DNS 服务器再将请求发送到网站所在的服务器。网站所在的服务器在收到请求并解析后，将网页的 HTML、JS、CSS 等文件返回给浏览器，再经过浏览器的解析加载后，用户才可以看到排版工整、有着各种特效的页面。而服务器发送给浏览器的没有被解析的源代码可能是如图 12-3 所示的样子。

通过上述分析我们不难想象，可以使用 urllib 库来模拟浏览器访问网站的过程，通过分析服务器返回的源代码的 URL 和媒体信息资源，来得到想要的信息。

URL，即统一资源定位符(Uniform Resource Locator)，也就是常说的网址，它简洁地表达了互联网上可得到的资源的位置，同时也指出了访问该资源的方法，是互联网上标准资源的地址。换言之，互联网上的每个文件都有一个唯一的 URL，它包含的信息指出了文

件的位置以及浏览器应该怎么处理它。

```
onmouseover="this.className='s_btn s_btn_on'" value="百度一下"/> </span> </
href="javascript:void(0)" title="上传图片，搜索相关信息"> <img class="st_ca
src="//img1.bdstatic.com/static/common/widget/shitu/images/camera_new_off_a
src="//img2.bdstatic.com/static/common/widget/shitu/images/camera_new_on_4e
息</div> <div class="st_tips_arrow_in" id="stTipArrowIn"></div> <div class=
style="display:none"> <div id="sthead"><span>识图</span><img id="sthelp" wi
src="//img2.bdstatic.com/static/common/widget/shitu/images/mark_b68ff2e.png
action="/pictureup/uploadshitu" method="post" name="form2"> <a id="uploadIm
nsclick="p=1811102&tn=index&event_type=shitu.search.click&pos=upload">本地_
id="dragtg" style="display:none;">提示：您也可以把图片拖到这里</div> <input
type="hidden"> <input name="fm" value="index" type="hidden"> </form> <form
method="get" name="form1"> <div id="sturl"> <span class="stuwr"> <input typ
class="stuurl" name="objurl"> </span> <span class="stsb"> <input type="subm
onmouseout="this.className='stsb2'" onmouseover="this.className='stsb2 stst
type="hidden"> <input name="rt" value="0" type="hidden"> <input name="rn" v
name="ct" value="1" type="hidden"> <input name="stt" value="0" type="hidder
```

图 12-3　浏览器未解析的源代码

URL 的格式由三部分组成。

(1) 第一部分是协议(或称为服务方式)。

(2) 第二部分是存有该资源的主机 IP 地址(有时也包括端口号)。

(3) 第三部分是主机资源的具体地址，如目录和文件名等。

爬虫爬取数据时必须有一个目标 URL 才可以获取数据，因此，URL 是爬虫获取数据的基本依据，准确地理解它的含义，对学习爬虫知识有很大的帮助。

12.4.3　urllib 库

Python 3 的 urllib 模块是一堆可以处理 URL 的组件集合。

urllib 模块包含了 4 个子模块：

- urllib.request
- urllib.error
- urllib.parse
- urllib.rebotparser

其中，主要使用 request 打开 URL 来访问网页。访问 URL 的代码如下：

```
urllib.request.urlopen(url, timeout)
```

其中的 url 可以是字符串格式的 URL 地址，也可以是 Request 对象。urlopen()函数会返回一个 Request 对象，使用.read()方法就可以读出网页信息了。

但是，使用.read()方法下载网络数据时，会因为对方网速慢、服务器超时等原因而导致“卡死”。解决这一问题的方法有多种，其中最简单的就是设置可选参数 timeout，例如用 timeout=10 可将超时时间设置为 10 秒。此外，通过 socket 模块的 setdefaulttimeout()函数也可以控制超时时间，例如，用如下代码也可将超时时间设置为 10 秒：

```
socket.setdefaulttimeout(10)
```

件的位置以及浏览器应该怎么处理它。

```
onmouseover="this.className='s_btn s_btn_on'" value="百度一下"/> </span> </
href="javascript:void(0)" title="上传图片，搜索相关信息"> <img class="st_ca
src="//img1.bdstatic.com/static/common/widget/shitu/images/camera_new_off_a
src="//img2.bdstatic.com/static/common/widget/shitu/images/camera_new_on_4e
息</div> <div class="st_tips_arrow_in" id="stTipArrowIn"></div> <div class=
style="display:none"> <div id="sthead"><span>识图</span><img id="sthelp" wi
src="//img2.bdstatic.com/static/common/widget/shitu/images/mark_b68ff2e.png
action="/pictureup/uploadshitu" method="post" name="form2"> <a id="uploadIm
nsclick="p=1811102&tn=index&event_type=shitu.search.click&pos=upload">本地_
id="dragtg" style="display:none;">提示：您也可以把图片拖到这里</div> <input
type="hidden"> <input name="fm" value="index" type="hidden"> </form> <form
method="get" name="form1"> <div id="sturl"> <span class="stuwr"> <input typ
class="stuurl" name="objurl"> </span> <span class="stsb"> <input type="subm
onmouseout="this.className='stsb2'" onmouseover="this.className='stsb2 stsb
type="hidden"> <input name="rt" value="0" type="hidden"> <input name="rn" v
name="ct" value="1" type="hidden"> <input name="stt" value="0" type="hidden
```

图 12-3　浏览器未解析的源代码

URL 的格式由三部分组成。

(1)　第一部分是协议(或称为服务方式)。

(2)　第二部分是存有该资源的主机 IP 地址(有时也包括端口号)。

(3)　第三部分是主机资源的具体地址，如目录和文件名等。

爬虫爬取数据时必须有一个目标 URL 才可以获取数据，因此，URL 是爬虫获取数据的基本依据，准确地理解它的含义，对学习爬虫知识有很大的帮助。

12.4.3　urllib 库

Python 3 的 urllib 模块是一堆可以处理 URL 的组件集合。

urllib 模块包含了 4 个子模块：

- urllib.request
- urllib.error
- urllib.parse
- urllib.rebotparser

其中，主要使用 request 打开 URL 来访问网页。访问 URL 的代码如下：

```
urllib.request.urlopen(url, timeout)
```

其中的 url 可以是字符串格式的 URL 地址，也可以是 Request 对象。urlopen()函数会返回一个 Request 对象，使用.read()方法就可以读出网页信息了。

但是，使用.read()方法下载网络数据时，会因为对方网速慢、服务器超时等原因而导致“卡死”。解决这一问题的方法有多种，其中最简单的就是设置可选参数 timeout，例如用 timeout=10 可将超时时间设置为 10 秒。此外，通过 socket 模块的 setdefaulttimeout()函数也可以控制超时时间，例如，用如下代码也可将超时时间设置为 10 秒：

```
socket.setdefaulttimeout(10)
```

【例 12-4】 urlopen 方法的使用：

```
import urllib.request as rq

uop = rq.urlopen("http://mini.eastday.com/", timeout=100)    #打开网页
data = uop.read(800)   #使用.read()方法获取服务器返回的内容，返回 800 个字符
print(data)
```

输出结果如下所示：

```
b'<!doctype html>\n<html lang="zh-cmn-Hans-CN">\n<head>\n    <meta charset
="utf-8" />\n    <meta name="renderer" content="webkit" />\n    <meta http
-equiv="X-UA-Compatible" content="IE=Edge,chrome=1" />\n    <META name="fi
letype" content="1">\n    <META name="publishedtype" content="1">\n    <ME
TA name="pagetype" content="2">\n    <META name="catalogs" content="toutia
o_PC">\n    <meta name="applicable-device" content="pc">\n    <link href="
/assets/images/favicon.ico" type="image/x-icon" rel="icon" />\n    <link r
el="canonical" href="http://mini.eastday.com/" />\n    <title>\xe5\xa4\xb4
\xe6\x9d\xa1\xe6\x96\xb0\xe9\x97\xbb_\xe4\xb8\x9c\xe6\x96\xb9\xe5\xa4\xb4\
xe6\x9d\xa1</title>\n    <meta name="keywords" content="\xe4\xb8\x9c\xe6\x
96\xb9\xe5\xa4\xb4\xe6\x9d\xa1,\xe5\xa4\xb4\xe6\x9d\xa1\xe6\x96\xb0\xe9\x9
7\xbb,\xe5\xa4\xb4\xe6\x9d\xa1,\xe4\xbb\x8a\xe6\x97\xa5\xe6\x96\xb0\xe9\x9
7\xbb\xe5\xa4\xb4\xe6\x9d\xa1,\xe5\xa4\xb4\xe6\x9d\xa1\xe7\xbd\x91,\xe5\xa
4\xb4\xe6\x9d\xa1\xe6\x96\xb0\xe9\x97\xbb,\xe4\xbb\x8a\xe6\x97\xa5\xe5\xa4
\xb4\xe6\x9d\xa1\xe6\x96\xb0\xe9\x97\xbb" />\n    <meta name="description"
content="\xe4\xb8\x9c\xe6\x96\xb9\xe5\xa4\xb4\xe6\x9d\xa1\xe7\xbd\x91 \xe4
\xb8\x9c\xe6\x96\xb9\xe7\xbd\x91 \xe6\x97\x97\xe4'
>>>
```

可以看到，返回的代码中汉字部分被解析为“\xe7\x99\”等字符，这是网络编码问题引起的，只需要将.read()方法改为.read().decode(“utf-8”)就可以解决这一问题。

网络编码问题解决后，实际运行结果如下：

```
<!doctype html>
<html lang="zh-cmn-Hans-CN">
<head>
    <meta charset="utf-8" />
    <meta name="renderer" content="webkit" />
    <meta http-equiv="X-UA-Compatible" content="IE=Edge,chrome=1" />
    <META name="filetype" content="1">
    <META name="publishedtype" content="1">
    <META name="pagetype" content="2">
    <META name="catalogs" content="toutiao_PC">
    <meta name="applicable-device" content="pc">
    <link href="/assets/images/favicon.ico" type="image/x-icon" rel="icon" />
    <link rel="canonical" href="http://mini.eastday.com/" />
    <title>头条新闻_东方头条</title>
    <meta name="keywords" content="东方头条,头条新闻,头条,今日新闻头条,头条网,头
条新闻,今日头条新闻" />
    <meta name="description" content="东方头条网 东方网 旗下《东方头条》是一款会
自动学习的资讯软件,它会分析你的兴趣爱好,为你推荐喜欢的内容,并且越用越懂你.就要你
好看,东方头条新闻网!" />
```

前面刚刚提到，urlopen(url, timeout)中的 url 还可以是一个 Request 对象。使用 Request 对象可以同时向服务器发送 URL、data、header 等内容。

建立 Request 对象的方法如下：

```
urllib.request.Request(url,data=None,headers={},origin req host=None,
unverifiable=False,method=None)
```

其中前面三个参数最为常用：url 为 URL 字符串；可选参数 data(UTF-8 编码格式)是向服务器传送的数据；headers 为头文件字典，主要用于设置模拟浏览器访问，headers 的主要参数如表 12-4 所示；后面三个参数很少使用，一般分别设置为 None、False、None。

表 12-4　headers 的主要参数

参　数	说　明
User-Agent	服务器或 Proxy 会通过该值来判断是否是浏览器发出的请求
Content-Type	在使用 REST 接口时，服务器会检查该值，用来确定 HTTP Body 中的内容该怎样解析。在使用服务器提供的 RESTful 或 SOAP 服务时，Content-Type 设置错误会导致服务器拒绝服务
application/xml	在 XML RPC，如 RESTful/SOAP 调用时使用
application/json	在 JSON RPC 调用时使用
application/x-www-form-urlencoded	浏览器提交 Web 表单时使用

Request 对象的主要方法如表 12-5 所示。

表 12-5　Request 对象的主要方法

方　法	说　明
request.full_url	request 对象的 URL
request.host	主机地址和主机端口号
request.data	将要向服务器发送的数据(如账号、密码等)
request.method	数据传输方式(POST 或 GET)
request.add_data(data)	向 request 对象中添加传输数据
request.add_header(key,val)	向 request 对象中添加头文件

【例 12-5】Request 对象使用示例：

```
import urllib.request as rq
import urllib.parse

headers = {'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows
NT)', 'Referer': 'http://www.zhihu.com/articles'}

values = {"username":"1016903103@qq.com","password":"XXXX"}
data = urllib.parse.urlencode(values)
url = "https://passport.csdn.net/account/login?
from=http://my.csdn.net/my/mycsdn"
request = rq.Request(url,data.encode(),headers)
response = rq.urlopen(request)
print(response.read().decode("utf-8"))
```

在上面的代码中，需要向服务器提交账号密码表单，在传递表单信息前，还需要使用 urllib.parse.urlencode()函数进行数据格式转换，然后再将格式转换后的数据传入 urlopen。

当然，在实际登录的过程中，可能还需要流水段字号或者验证码等，因而，运行上面的代码并不会真的登录成功。

如果读者有兴趣，可以阅读相关资料，学习相关的操作。

12.5 案例实训：设计获取网站热点要闻的网络爬虫程序

前面介绍了与网络爬虫相关的知识，下面将以本章前述内容为基础，并结合前面几章所学的大部分知识，实现一个用于获取“东方头条”网站热点要闻的网络爬虫。整个爬虫程序的执行流程如下。

(1) 登录“东方头条”网站，并获取服务器返回的网页源代码。

(2) 获取源代码中所有热点新闻的 URL。

(3) 爬取热点新闻的 URL 并从中提取出所有的新闻图片。

(4) 对于包含“下一页”链接的热点新闻，再次提取所有子页 URL，并提取信息。

(5) 保存图片文件到本地文件夹。

(6) 保存文字信息到本地形成文本文档。

(7) 重复(3)~(6)步，直到所有热点新闻的 URL 都被访问完毕。

下面先讲解三个关键步骤，然后给出爬虫程序的完整代码。

1. 打开《东方头条》网站，通过正则表达式匹配热点新闻的 URL

前面已经介绍过打开一个网站并返回网站源代码的方法：只需要使用.urlopen()函数打开网页，并通过.read().decode(“utf-8”)方法，即可返回一个 utf-8 格式的网页源代码文件。具体代码如下：

```
import urllib.request as rq

uop = rq.urlopen("http://mini.eastday.com/", timeout=100)    #打开起始网页
data = uop.read().decode("utf-8")
print(data)
```

但从图 12-3 可以看到，返回的源代码是杂乱的，想从中提取出热点新闻区域是很麻烦的。这时，如果使用的浏览器支持审查元素功能，也可以利用这一功能帮助我们快速地确定目标元素的位置，如图 12-4 所示。



图 12-4 审查元素功能

使用审查元素的功能，可以看到如图 12-5 所示的格式十分工整的 HTML 代码。需要的新闻内容包含在<div id="J_hot_news" class="hot-news clearfix">标签中，再次通过单击标签前面的三角就可以很清晰地看到，需要提取的 URL 信息和新闻标题名在子标签<a>中和<a>与之间。通过使用第 2 章 2.8 节介绍的正则表达式匹配方法，可以从中提取信息。



图 12-5　显示代码

通过实验，代码如下：

```
import urllib.request as rq
import re
uop = rq.urlopen("http://mini.eastday.com/", timeout=100)    #打开起始网页
data = uop.read().decode("utf-8")
pattern = re.compile('<a class="title-lg" href="(.+?)".*?title="(.+?)">')
#提取热点要闻的所有新闻 URL
items = re.findall(pattern, data)
for item in items:
    print(str(item[1])+" : "+str(item[0]))
```

其中，正则表达式中的第一个(.+?)中保存的是 URL 地址，新闻的标题内容保存在第二个(.+?)中，对应于匹配结果中的 item[0]和 item[1]，输出结果如图 12-6 所示。经人工对比匹配结果和网页内容，确定所有的热点新闻都已被匹配到了。

```
今年GDP预期增长6.5% 解决群众普遍关心的问题 : //mini.eastday.com/a/170306011009603.html
北京地铁辱骂他人男子仅17岁 或将逃过实际处罚 : //mini.eastday.com/a/170305215850805.html
韩美启动最大规模联合军演 &quot;萨德&quot;参与模拟军演 : //mini.eastday.com/a/170306070253127.html
恒大战国安连续7场不败 卫冕之旅从死敌身上开启 : //mini.eastday.com/a/170306081806592.html
政府工作报告传递2017年中国发展八大信号 : //mini.eastday.com/a/170306074104071.html
境外媒体称台湾紧盯大陆西太军演：忧大陆军力增强 : //mini.eastday.com/a/170306083911989.html
张国立在两会提了个建议 被台湾记者堵厕所质问 : //mini.eastday.com/a/170306070231435.html
澳门有“旅游警察”了：驻守观光区 保障出行安全 : //mini.eastday.com/a/170306011043728.html
沪上旧书业“换一种活法”：所有人都是旧书经营者 : //mini.eastday.com/a/170306051046482.html
成都发现“地下青铜器宝库” 源于春秋战国时期 : //mini.eastday.com/a/170306011048234.html
```

图 12-6　匹配后的结果

注意： 作为网站的开发者和维护者，并不希望他人编写爬虫程序随意爬取网站的内容，所以网站可能会定期或不定期地变换网页源文件格式，甚至会更换编码方式。这对编写爬虫程序而言的确是一种挑战！有鉴于此，实际爬取网页时，尚需根据实际情况重新构造用于提取文本的正则表达式，否则可能达不到网页爬取的目的。换言之，本书提供的程序在调试时能够正常爬取网站内容，但并不意味着网站更新后一定能够正常爬取，有可能需要重新编写正则表达式，甚至尝试更换一下编码方式。本节程序在调试过程中已遇到了这样的难题。

2. 打开热点新闻 URL，从中提取信息

在图 12-6 中，可以发现所有的 URL 都是不全的，都缺少前面的“http:”这可能是网站编写时的习惯，需要及时补全，以免无法访问。实现此功能的具体代码如下：

```
def info extract(data):
    #信息提取函数，用来提取出网页中的文本和图片
    pattern= re.compile(
        '<div class="gg detail cnt clearfix" id="dsp btxf">(.*?)<div
class="gg item bomttom cnt" id="dsp left2">')
    items = re.findall(pattern, data)    #在网页中提取包含信息的最小范围
    if items != []:
        pattern jpg1 = re.compile('<img.*?src="(.*?)".*?>')    #提取图片
        pattern jpg2 = re.compile("<img.*?src='(.*?)'>")    #提取图片
        jpg = re.findall(pattern jpg1, str(items[0]))
        jpg += re.findall(pattern jpg2, str(items[0]))
    else:
        jpg = []
    pattern text = re.compile("<p.*?>([^<].*?)</p>|<span.*?>(.*?)</span>")
#提取文本
    text = re.findall(pattern text, str(items))
    return (text,jpg)    #使用一个元组返回找到的图片和文本列表

for item in items:
    text = []
    URL = "http:" + str(item[0])  #完整的 URL 格式
    uop = rq.urlopen(URL, timeout=100)    #打开新闻 URL 开始提取信息
    data = uop.read().decode("utf-8")
    pattern next = re.compile('<a href="(.*?)">\d</a>')#提取新闻的下一页 URL
    next = re.findall(pattern next, data)
    response = info extract(data) #调用 info extract()函数提取新闻网页中的信息
    text = text + response[0]
    jpg = jpg+response[1]

    if len(next) != 0:    #如果下一页 URL 的提取结果不为空，继续提取信息
        for i in range(0, len(next)):
            next url = str(URL).replace(".html", "-"+str(i+2)+".html")
              #完整子页 URL
            uop = rq.urlopen(next url, timeout=100)
            data = uop.read().decode("utf-8")
```

```
            response = info extract(data)    #获取子页信息
            text = text + response[0]
            jpg = jpg + response[1]
    print(text)
    print(jpg)
```

下面对上述代码做几点说明。首先通过.urlopen()方法打开 URL。因为有的新闻内容较多，可能存在子页，所以还需提取出所有的子页 URL。将提取信息的代码定义为函数 info_extract(data)，该函数将匹配到的文本内容和图片以元组的形式返回。然后，通过 text、jpg 列表收集每条热点新闻子页的所有文本信息和图片地址。最后，加入一个 for 循环处理所有的 URL。

3. 保存数据到本地

至此，所有热点新闻的数据均已收集到了，最后一步是将所有数据保存到本地。相关操作在第 5 章文件与异常处理中已介绍过，代码如下：

```
import os
import urllib.request

def file name(Title):
    #文件名过滤函数，滤掉标题中不合理的字符
    Title = str(Title).replace("?", "")
    Title = str(Title).replace("*", "")
    Title = str(Title).replace(":", "")
    Title = str(Title).replace("<", "")
    Title = str(Title).replace(">", "")
    Title = str(Title).replace("|", "")
    Title = str(Title).replace("&quot;", "")
    return Title

path = os.getcwd()    #查看当前路径
title = file name(item[1])  #设置保存的文件名
add = path + "\\" + title  #设置保存路
isExists = os.path.exists(add)
if not isExists:  #在当前路径创建文件夹，如果没有同名文件夹则新建
    os.makedirs(add)
f = open(add + "\\" + title + ".txt", 'w')    #保存文本
for i in text:
    for j in i:
        f.write(str(j))
f.close()

n = 0
for image in images:
    try:  #因为图片可能打不开，所以使用 try 使程序继续执行
        u = urllib.request.urlopen("http:" + str(image), timeout=10)
          #访问图片
        file = u.read()
        f = open(add + "\\" + str(n) + ".jpg", 'wb')    #保存图片
```

```
        f.write(file)
        f.close()
        n = n + 1
    except Exception as e:
        pass
```

程序解释： 首先使用 os.getcwd()方法获取当前的文件路径，将新闻的标题作为文件夹和文本文档的文件名，使用一个 file_name()函数规范化文件名。然后，使用 os.makedirs()函数按照新闻标题建立文件夹，并将文本文档和图片保存到该文件夹下。

4. 网络爬虫程序的完整代码

最后附上网络爬虫程序的完整代码，以供读者参考。

用于爬取“东方头条”热点新闻的简单网络爬虫程序：

```
import urllib.request as rq
import urllib.parse
import re
import os

def info extract(data):
    #信息提取函数，用来提取出网页中的文本和图片
    pattern= re.compile(
        '<div class="gg detail cnt clearfix" id="dsp btxf">(.*?)<div
class="gg item bomttom cnt" id="dsp left2">')
    items = re.findall(pattern, data)    #在网页中提取包含信息的最小范围
    if items != []:
        pattern jpg1 = re.compile('<img.*?src="(.*?)".*?>')    #提取图片
        pattern jpg2 = re.compile("<img.*?src='(.*?)'>")    #提取图片
        jpg = re.findall(pattern jpg1, str(items[0]))
        jpg += re.findall(pattern jpg2, str(items[0]))
    else:
        jpg = []
    pattern text = re.compile("<p.*?>([^<].*?)</p>|<span.*?>(.*?)</span>")
#提取文本
    text = re.findall(pattern text, str(items))
    return (text,jpg)    #使用一个元组返回找到的图片和文本列表

def image save(images):
    #图片保存函数
    n=0
    for image in images:
        try:  #因为图片可能打不开，所以使用 try 使程序继续执行
            u = urllib.request.urlopen("http:" + str(image), timeout=10)
#访问图片
            file = u.read()
            f = open(add + "\\" + str(n) + ".jpg", 'wb')    #保存图片
            f.write(file)
            f.close()
            n = n + 1
        except Exception as e:
```

```
            pass

def file name(Title):
    #文件名过滤函数，滤掉标题中不合理的字符
    Title = str(Title).replace("?", "")
    Title = str(Title).replace("*", "")
    Title = str(Title).replace(":", "")
    Title = str(Title).replace("<", "")
    Title = str(Title).replace(">", "")
    Title = str(Title).replace("|", "")
    Title = str(Title).replace("&quot;", "")
    return Title

#驱动
if   name   == '  main  ':
    uop = rq.urlopen("http://mini.eastday.com/", timeout=100) #打开起始网页
    data = uop.read().decode("utf-8")
    pattern = re.compile('<a class="title-lg"
href="(.+?)".*?title="(.+?)">')    #提取热点要闻的所有新闻 URL
    items = re.findall(pattern, data)
    path = os.getcwd()    #查看当前路径

    for item in items:
        text = []
        jpg = []
        URL = "http:" + str(item[0])  #完整的 URL 格式
        title = file name(item[1])  #设置保存的文件名
        add = path + "\\" + title  #设置保存路

        isExists = os.path.exists(add)
        if not isExists:    #在当前路径创建文件夹，如果没有同名文件夹则新建
            os.makedirs(add)

        uop = rq.urlopen(URL, timeout=100)    #打开新闻 URL 开始提取信息
        data = uop.read().decode("utf-8")
        data = str(data).replace("\n", "")
        pattern next = re.compile('<a href="(.*?)">\d</a>')
           #提取新闻的下一页 URL
        next = re.findall(pattern next, data)
        response = info extract(data)
           #调用 info extract()函数提取新闻网页中的信息
        text = text + response[0]
        jpg = jpg + response[1]

        if len(next) != 0:    #如果下一页 URL 的提取结果不为空，继续提取信息
            for i in range(0, len(next)):
                next url = str(URL).replace(".html", "-"+str(i+2)+".html")
                   #完整子页 URL
                uop = rq.urlopen(next url, timeout=100)
                data = uop.read().decode("utf-8")
```

```
            data = str(data).replace("\n", "")
            response = info extract(data)    #获取子页信息
            text = text + response[0]
            jpg = jpg + response[1]

    f = open(add + "\\" + title + ".txt", 'w')    #保存文本
    for i in text:
        for j in i:
            f.write(str(j))
    f.close()

    image save(jpg)    #保存图片
    print("信息保存成功: " + title)
print("东方头条热点要闻爬取完毕。")
```

运行结果如图 12-7 所示，收集到的数据已被成功地保存到本地文件夹中，如图 12-8、图 12-9 所示。

```
信息保存成功：建军90周年大会举行 习近平出席并发表讲话
信息保存成功：31省区市上半年GDP出炉 各省区市均富可敌国
信息保存成功：7省份环保问题清单出炉 多地假装治污被点名
信息保存成功：这100人进入秦城监狱面试名单 有岗位仅限女性
信息保存成功：亚裔男子在中国驻洛杉矶总领馆开枪 随后自杀
信息保存成功：朝鲜谴责美国会通过新一轮对朝制裁议案无耻
信息保存成功：土耳其政变最大规模审判近500名头目被押出场
信息保存成功：中国在东海进行移动式挖掘船作业 日方提抗议
信息保存成功：瓜农制止偷窃被刺死获荣誉称号 家属获奖10万
信息保存成功：深大失联女生在港盗窃被捕 辅导员谁都会犯错
信息保存成功：北京现灵魂画手20元完全不像 创作不看人
东方头条热点要闻爬取完毕。
```

图 12-7　程序运行日志



图 12-8　保存爬取数据的文件夹



图 12-9　爬取的图片文件

本 章 小 结

本章主要介绍了如何使用 Python 进行网络编程。因为 socket 是很多其他通信类库的基础，也是网络通信的基础，所以本章先以一个端口扫描器为例讲解了 socket 的使用。之后，通过一个简单的网络爬虫例子讲解了 urllib 库的使用。使用 urllib 库，可以帮助我们快速地抓取网页信息，大大地提高信息获取效率。本章最后通过一个获取“东方头条”网站热点要闻的网络爬虫程序，将前面几章所学的内容进行了回顾和综合应用，旨在通过一个案例实训帮助读者学以致用。

习 题

1. 填空题

(1) Internet 采用一种全局通用的地址格式，为网络中的每一台主机都分配一个唯一的地址，该地址称为(　　　)。

(2) Internet 使用(　　　)来管理计算机域名与 IP 地址的对应关系。

(3) IP 地址用来标识 Internet 上的主机，而位于 Internet 主机上的资源(如各种文档、图像等)则通过(　　　)来标识。

(4) TCP/IP 协议的传输层包含两个传输协议：面向连接的(　　　)和非面向连接的(　　　)。

(5) 创建服务器端 socket 对象并绑定到 IP 地址后，可以使用(　　　)和(　　　)对象方法进行侦听和接收连接。

(6) 客户端 socket 对象通过(　　　)方法尝试建立到服务器 socket 对象的连接。

2. 选择题

(1) TCP/IP 协议参考模型把所有的 TCP/IP 系列协议归类到 4 个抽象层中，这 4 个层按照由高到低分别为(　　)。

A. 应用层、传输层、网络层和链路层

B. 链路层、网络层、传输层和应用层

C. 应用层、网络层、传输层和链路层

D. 链路层、传输层、网络层和应用层

(2) 一般情况下，Web 服务器可能位于计算机的 TCP(　　)端口。

A. 25　　B. 21　　C. 80　　D. 102

(3) 在 urllib 库中，我们常用(　　)子模块打开 URL 来访问网页。

A. parse　　B. rebotparser　　C. error　　D. request

3. 问答题

(1) 简述 socket 通信的连接过程。

(2) urllib 模块包含哪 4 个子模块，简述其分别实现什么功能。



附录 A

Python 关键字

Python 3 中的关键字共计 33 个，它们是：

```
False       class       finally     is          return
None        continue    for         lambda      try
True        def         from        nonlocal    while
and         del         global      not         with
as          elif        if          or          yield
assert      else        import      pass
break       except      in          raise
```

各关键字的详解如表 A-1 所示。

表 A-1　Python 关键字详解

<table>
<tr><th>关 键 字</th><th>详　解</th></tr>
<tr><td>False</td><td>布尔类型的值，表示假，与 True 相反</td></tr>
<tr><td>class</td><td>定义类的关键字</td></tr>
<tr><td>finally</td><td>在异常处理的时候添加，有了它，程序始终要执行 finally 里面的程序代码块，如：<pre>class MyException(Exception):pass
try:
    #some code here
    raise MyException
except MyException:
    print “MyException encoutered”
finally:
    print “Arrive finally”</pre></td></tr>
<tr><td>is</td><td>Python 中的对象包含三要素：id、type、value，其中 id 用来唯一标识一个对象，type 标识对象的类型，value 是对象的值。is 判断的是 a 对象是否就是 b 对象，它是通过 id 来判断的；==判断的是 a 对象的值是否和 b 对象的值相等，它是通过 value 来判断的。如：<pre>>>> a = 1
>>> b = 1.0
>>> a is b
False
>>> a == b
True
>>> id(a)
12777000
>>> id(b)
14986000</pre></td></tr>
<tr><td>return</td><td>return 语句用来从一个函数返回，即跳出函数。也可选从函数返回一个值</td></tr>
<tr><td>None</td><td>None 是一个特殊的常量。None 和 False 不同，None 不是 0，也不是空字符串。None 和任何其他的数据类型比较，永远返回 False。None 有自己的数据类型 NoneType。可以将 None 复制给任何变量，但是不能创建其他 NoneType 对象。<pre>>>> type(None)
<class ‘NoneType’>
>>> None == 0
False
>>> None == ‘’
False
>>> None == None
True
>>> None == False
False</pre></td></tr>
</table>

续表

关 键 字	详　解
continue	continue 语句被用来告诉 Python 跳过当前循环体中的剩余语句，然后继续进行下一轮循环
for	for...in 是另外一个循环语句，它在一序列的对象上递归，即逐一使用序列中的每个项目

lambda	匿名函数是个很时髦的概念，提升了代码的简洁程度。如：
	`g = lambda x: x*2`
	`g(3)`

try	可以使用 try...except 语句来处理异常。我们把通常的语句放在 try 块中，而把错误处理语句放在 except 块中
True	布尔类型的值，表示真，与 False 相反

def	`#定义函数`
	`def hello():`
	`    print('hello, Python')`
	`#调用函数`
	`hello()`
	`hello, Python`

from	在 Python 中用 import 或者 from...import 来导入相应的模块

nonlocal	nonlocal 关键字用来在函数或其他作用域中使用外层(非全局)变量，如：
	`def make counter():`
	`    count = 0`
	`    def counter():`
	`        nonlocal count`
	`        count += 1`
	`        return count`
	`    return counter`
	
	`def make counter test():`
	`  mc = make counter()`
	`  print(mc())`
	`  print(mc())`
	`  print(mc())`

while	while 语句允许重复执行一块语句。while 语句是所谓循环语句的一个例子。while 语句有一个可选的 else 从句
and	逻辑“与”，和 C 的&&一样

del	del 用于 list 列表操作，删除一个或者连续几个元素。如：
	`a = [-1, 3,'aa', 85] #定义一个 list`
	`del a[0] #删除第 0 个元素`
	`del a[2:4] #删除从第 2 个元素开始，到第 4 个为止的元素，包括头但不包括尾`

global	定义全局变量
not	逻辑“非”，和 C 的 ！一样

with	with 是 Python 2.5 以后才有的，它实质是一个控制流语句，with 可以用来简化 try-finally 语句。它的主要用法是实现一个类__enter__()和__exit__()方法，如：
	`class controlled execution:`
	`    def  enter  (self):`
	`        set things up`
	`        return thing`
	`    def  exit  (self, type, value, traceback):`
	`        tear thing down`
	`with controlled_execution() as thing:`
	`    some code`

续表

关 键 字	详　解
as	结合 with 使用
elif	和 if 配合使用
if	if 语句用来检验一个条件，如果条件为真，运行一语句块(称为 if 块)，否则处理另外一语句块(称为 else 块)。else 子句是可选的
or	逻辑“或”，和 C 的 || 一样

yield	yield 是关键字，用起来像 return，yield 在告诉程序，要求函数返回一个生成器，如：
	`def createGenerator():`
	`mylist = range(3)`
	`for i in mylist:`
	`yield i*i`

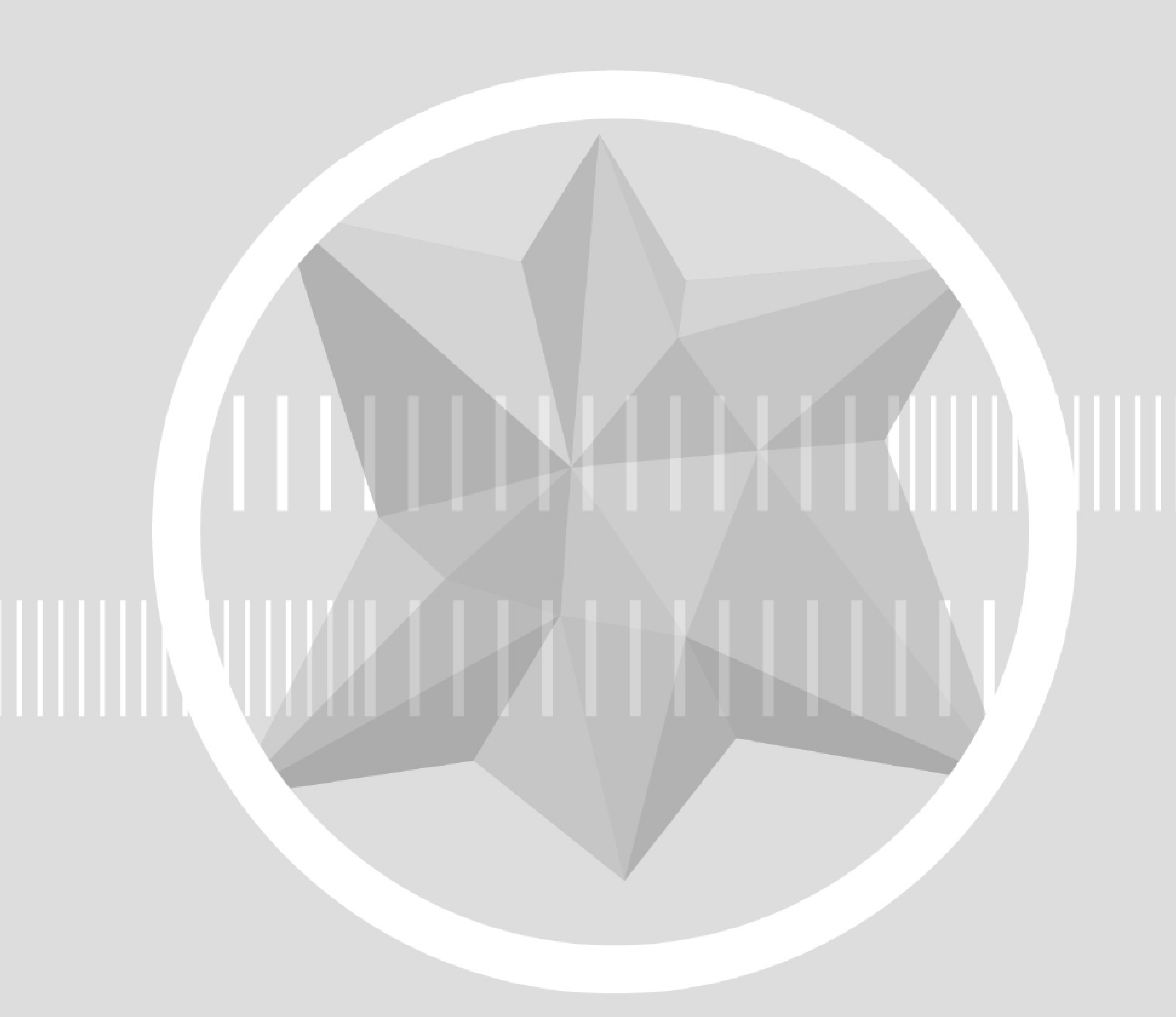
assert	断言，这个关键字用来在运行中检查程序的正确性，和很多其他语言是一样的作用。如： `assert len(mylist) >= 1`
else	与 if 关键字配合使用，　但 else 子句不是必需的

import	在 Python 中用 import 或者 from...import 来导入相应的模块，如：
	`from sys import *`
	`print('path:',path)`

pass	pass 的意思是什么都不要做，作用是为了弥补语法和空定义上的冲突，其好处体现在代码的编写过程之中，比如你可以先写好软件的整个框架，然后再填好框架内具体函数和 class 的内容，如果没有 pass 编译器会报一堆的错误，让整个开发过程很不流畅，如：
	`def f(arg): pass    #a function that does nothing (yet)`
	`class C: pass      #a class with no methods (yet)`

break	break 语句是用来终止循环语句的，即哪怕循环条件没有成为 False 或序列还没有被完全递归，也停止执行循环语句。 一个重要的注释是，如果从 for 或 while 循环中终止，任何对应的循环的 else 块将不执行
except	使用 try 和 except 语句来捕获异常
in	for...in 是另外一个循环语句，它在一序列的对象上循环，即逐一使用队列中的每个项目

raise	Python 的 raise 和 Java 的 throw 很类似，都是抛出异常。如：
	`class MyException(Exception):pass`
	`try:`
	`    #some code here`
	`    raise MyException`
	`except MyException:`
	`    print "MyException encoutered"`
	`finally:`
	`    print "Arrive finally"`



附录 B

其他常用功能

Python 的一大优势便是其可扩展性，在此基础上衍生出了数量庞大的第三方扩展库，因而大大增强了 Python 的功能。

表 B-1 所列的是前面各章未曾提到的其他一些常用功能。

表 B-1 Python 的其他常用功能

功　能	第三方扩展库	说　明	下载地址
多线程处理	eventlet	使用 green threads 概念，资源开销很少	http://eventlet.net/
界面编程	wxPython	其消息机制与 MFC 颇为相似。入门简单，适合快速开发应用	http://www.wxpython.org/
可执行文件生成	py2exe (Python to EXE)	将 Python 脚本文件打包成 Windows 下的 exe 文件	http://www.py2exe.org/
图像处理	PIL(Python Image Library)	图像增强、滤波、几何变换以及序列图像处理，支持数十种图像格式。可直接载入图像文件、读取处理过的图像，或通过抓取方法得到的图像	http://pythonware.com/products/pil/
系统资源使用信息获取	pstuil	跨平台地获取和控制系统的进程，读取系统的 CPU 占用、内存占用以及磁盘、网络、用户等信息	http://code.google.com/p/psutil/
计算机视觉	OpenCV	具有图像处理和计算机视觉方面的很多通用算法，可用于人脸识别、物体识别、运动跟踪、机器视觉、动作识别、运动分析等	http://opencv.org/downloads.html
三维可视化	VTK	三维计算机图形学、图像处理和可视化	http://vtk.org/get-software.php
医学图像处理	ITK	用于处理医学图像，有丰富的图像分割与配准的算法程序	http://www.itk.org/HTML/Download.htm
开源数据库连接	MySQLdb	对开源数据库 MySQL 的支持	http://sourceforge.net/projects/mysql-python

参 考 文 献

[01] 麦金尼. 利用 Python 进行数据分析[M]. 唐学韬，等，译. 北京：机械工业出版社，2014.
[02] 张良均. Python 数据分析与挖掘实战[M]. 北京：机械工业出版社，2016.
[03] 巴里. Head First Python(中文版)[M]. 林琪，郭静，等，译. 北京：中国电力出版社，2012.
[04] 奥科罗. Python 绝技：运用 Python 成为顶级黑客[M]. 崔孝晨，武晓音，等，译. 北京：电子工业出版社，2016.
[05] 伊夫 • 希尔皮斯科. Python 金融大数据分析[M]. 姚军，译. 北京：人民邮电出版社，2015.
[06] 赫特兰. Python 基础教程[M]. 2 版. 司维，曾军崴，谭颖华，译. 北京：人民邮电出版社，2010.
[07] 刘浪，郭红涛，于晓强，等. Python 基础教程[M]. 北京：人民邮电出版社，2015.
[08] 萨卡尔. Python 网络编程攻略[M]. 安道，译. 北京：人民邮电出版社，2014.
[09] 董付国. Python 程序设计[M]. 北京：清华大学出版社，2015.
[10] 冯林. Python 程序设计与实现[M]. 北京：高等教育出版社，2015.
[11] 施奈德. Python 程序设计[M]. 车万翔，译. 北京：机械工业出版社，2016.
[12] 塞奇威克. 程序设计导论：Python 语言实践[M]. 江红，余青松，译. 北京：机械工业出版社，2016.
[13] 刘卫国. Python 程序设计教程[M]. 北京：北京邮电大学出版社，2016.
[14] 刘卫国. Python 语言程序设计[M]. 北京：电子工业出版社，2016.
[15] 周元哲. Python 程序设计基础[M]. 北京：清华大学出版社，2015.
[16] 宗成庆. 统计自然语言处理[M]. 北京：清华大学出版社，2013.
[17] 李航. 统计学习方法[M]. 北京：清华大学出版社，2012.
[18] 赵端阳，左伍衡. 算法分析与设计：以大学生程序设计竞赛为例[M]. 北京：清华大学出版社，2012.
[19] 裘宗燕. 数据结构与算法：Python 语言描述[M]. 北京：机械工业出版社，2016.
[20] 嵩天，黄天羽，礼欣. 程序设计基础：Python 语言[M]. 北京：高等教育出版社，2014.
[21] Eli Bressert. SciPy and NumPy: An Overview for Developers[M]. O’Reilly Media，2012.
[22] 谢希仁. 计算机网络[M]. 5 版. 北京：电子工业出版社，2008.
[23] 库罗斯，(美)罗斯. 计算机网络：自顶向下方法[M]. 6 版. 陈鸣，译. 北京：机械工业出版社，2014.
[24] 王达. 深入理解计算机网络[M]. 北京：机械工业出版社，2014.
[25] Gary. Wrigh，W. Richard Stevens. TCP/IP 详解[M]. 范建华，译. 北京：机械工业出版社，2000.
[26] 杨海霞，相洁，南志红. 数据库原理与应用[M]. 北京：人民邮电出版社，2013.
[27] 肖文鹏. 用 C 语言扩展 Python 的功能. http://www. ibm. com/developerworks/cn/linux/l-pythc/ [2003-02-03]
[28] Python 基础教程. https://www.runoob.com/python/python-tutorial.html [2017-07-10]
[29] Python 3 教程. https://www.runoob.com/python3/python3-tutorial.html [2017-07-10]
[30] 自强学堂. http://www.ziqiangxuetang.com/python3/python3-data-type.html [2017-07-10]
[31] 简书. http://www.jianshu.com/p/5383e4ba1318 [2015-11-04]
[32] 暮夏. http://www.muxia.org/2066.html [2012-08-27]

[33] 红黑联盟. http://www.2cto.com/kf/201602/491263.html [2016-02-29]

[34] 码农网. http://www.codeceo.com/article/python-regular-expressions-re-module.html [2016-12-27]

[35] Chinaunix.http://blog.chinaunix.net/uid-200142-id-4022131.html [2013-12-04]

[36] 玩蛇网. http://www.iplaypy.com/data/ [2017-06-30]

[37] Python 基础教程. https://www.runoob.com/python/python-mysql.html [2017-07-06]

[38] Scientific Computing Tools for Python.http://www.SciPy.org/ [2017-07-10]

[39] 脚本之家. http://www.jb51.net/list/list_97_1.htm [2017-06-19]

[40] Raymond Eric S. How to Become a Hacker. https://wenku.baidu.com/view/10e85e4ae45c3b3567ec8b8d.html? re=view[2011-03-19]

[41] PEP8 Python 编码规范. https://wenku.baidu.com/view/0d9535d8a300a6c30d229fc4.html.[2015-12-25]

[42] 《Python 之禅》中对于 Python 编程过程中的一些建议. http://www.jb51.net/article/63423.htm.[2015-04-03]

[43] 董付国. Python 可以这样学[M]. 北京：清华大学出版社，2017.